Imperial College of Science, Technology and Medicine
Department of Computing

Foundations for Behavioural Model Elaboration Using Modal Transition Systems

PhD Thesis

Dario Fischbein

Supervisor: Sebastian Uchitel

Submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College.

November 2011

Declaration of Originality

I hereby declare that the material presented in this thesis is my own, except where otherwise acknowledged and appropriately referenced.

Abstract

Modal Transition Systems (MTS) are an extension of Labelled Transition Systems (LTS) that have been shown to be useful to reason about system behaviour in the context of partial information. MTSs distinguish between required, proscribed and unknown behaviour and come equipped with a notion of refinement that supports incremental modelling where unknown behaviour is iteratively elaborated into required or proscribed behaviour.

A particularly useful notion in the context of software and requirements engineering is that of "merge". Merging two consistent models is a process that should result in a minimal common refinement of both models where consistency is defined as the existence of one common refinement. One of the current limitations of MTS merging is that a complete and correct algorithm for merging has not been developed. Hence, an engineer attempting to merge partial descriptions may be prevented to do so by overconstrained algorithms or algorithms that introduce behaviour that does not follow from the partial descriptions being merged. In this thesis we study the problems of consistency and merge for the existing MTSs semantics - strong and weak semantics - and provide a complete characterization of MTS consistency as well as a complete and correct algorithm for MTS merging using these semantics.

Strong and weak semantics require MTS models to have the same communicating alphabet, the latter allowing the use of a distinguished unobservable action. In this work we show that the requirement of fixing the alphabet for MTS semantics and the treatment of observable actions are limiting if MTSs are to support incremental elaboration of partial behaviour models. We present a novel observational semantics for MTS, branching alphabet semantics, inspired by branching LTS equivalence, which supports the elaboration of model behaviour including the extension of the alphabet of the system to describe behaviour aspects that previously had not been taken into account. Furthermore, we show that some unintuitive refinements allowed by weak semantics are avoided, and prove a number of theorems that relate branching refinement with alphabet refinement and consistency. These theorems, which do not hold for other semantics, support the argument for considering branching alphabet as a sound semantics to support behaviour model elaboration.

To Deby and Luca with love.

Acknowledgements

My most sincere thanks to my supervisor Sebastian Uchitel for his expert guidance throughout my PhD. His technical expertise and encouragement along the way have been vital to complete this work. I am also very grateful to him and Jeff Kramer for offering me the opportunity and ongoing support to complete this PhD.

I wish to thank Michael Huth and Jeff Magee for all the technical discussions and feedback throughout these years. Engaging in such discussions with them has always been quite an enjoyable challenge. I would also like to thank Victor Braberman not only for his advice, collaboration and for acting as a co-supervisor, but also for being instrumental in the process of continuing my studies in the UK.

Part of the work on this thesis has been motivated during some discussions with Marsha Chechik and Greg Brunet during my first year at Imperial, so I would like to thank both of them. I am also grateful to Nicolás D'Ippolito for his collaboration on the development of the MTSA tool presented in this thesis. I gladly acknowledge the financial support of the ERC StG PBM FIMBSE and the Department of Computing at Imperial College.

I would like to thank my family for all their love and encouragement. I am indebted to my parents for their guidance and for supporting me in all my pursuits.

My most special thank you to my beloved wife, Deby, for bringing joy to my life. I could not have done this thesis without her unconditional support and encouragement.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

The requirements and design of software systems are amenable to analysis through the construction of behaviour models, that is, formal operational descriptions of the intended system behaviour. This corresponds to the traditional engineering approach to construction of complex systems.

The major advantage of using models is that they can be studied to increase confidence in the adequacy of the product to be built. In particular, behaviour models used to describe software systems can be analysed and mechanically checked for properties in order to detect design errors early in the development process and allow less costly fixes [17].

Widely adopted formal methods that have proved to be successful in achieving these goals include Labelled Transition Systems (LTSs) [58] and Communicating Sequential Processes (CSP) [44]. These methods have been applied to the analysis and specification of distributed systems [16, 36, 69, 78] and security protocols [81, 68], among other applications.

The use of models as an integral part of the software development process is at the core of Model Driven Software Engineering (MDSE) [80, 8], an alternative approach to the more traditionally used techniques in the industry. This methodology proposes a shift from a code-centric towards a model-centric paradigm for software development.

Although there are clear advantages that stem from the use of an MDSE approach, the adoption of such methodologies by practitioners has been slow. This is in part due to a mismatch between most widely adopted software development techniques and a fundamental characteristic of traditional behaviour models.

On one side, as part of the essence of widely used iterative and incremental software development processes, the available system descriptions tend to be of a partial nature leaving some aspects of the desired behaviour undefined until a more advanced stage of the process is reached. Consequently, when the advantages of constructing models are more rewarding the complete system description is not available.

On the other side, semantics of traditional behaviour models such as LTSs assume a complete description of the system behaviour up to some level of abstraction, and hence cannot support reasoning in the presence of partial behavioural information. To support such reasoning, a behaviour model should allow at design time unknown aspects of system behaviour to be modelled. Moreover, it should provide a notion of elaboration of partial descriptions into more comprehensive ones.

Operational models that allow explicit modelling of unknown aspects of system behaviour are referred to as partial behavioural models. A number of such modelling formalisms exist, including Partial Labelled Transition Systems (PLTSs) [85], multi-valued state machines [23], Mixed Transition Systems [22], multi-valued Kripke structures [35, 11, 14]), and Modal Transition Systems (MTSs) [66].

Using partial behavioural models we can describe what is already known about the desired behaviour of the system at an early stage of the software development process, and analyse it in spite of not having complete knowledge of the expected system behaviour. In particular, we can distinguish between positive, negative, and unknown behaviours: positive behaviour refers to the behaviour that the system is expected to exhibit, negative behaviour refers to the behaviour that the system is expected to never exhibit, and unknown behaviour could become positive or negative, but the choice has not yet been made.

Further on, once all behavioural information is available, we will need a notion of conformance which defines whether a given non-partial behaviour model conforms to the required behaviour described in the partial model. If this is the case, we say that the total model is an implementation of the partial model. Therefore, a partial behaviour model can be thought of as specifying a set of possible implementations. These implementations arise from making different decisions on those aspects in which the model was partial. Moreover, if the notion of conformance can be shown formally to preserve certain properties, then all analysis previously done over the partial models will also be valid for the implementation.

Refinement [66] is a generalization of the notion of implementation that allows us to establish when one partial model conforms to another one. Intuitively, model

N is a refinement of model M if and only if the set of implementations of model N is a subset of the implementations of M. This notion allows us to start with an immature model and gradually and repeatedly evolve it while new knowledge is gathered until the complete system description is reached.

Another particularly useful notion in the context of software and requirements engineering is that of merge [66, 90]. Merging of operational behaviour models is similar to conjunction of declarative descriptions. The implementations described by a merge are those that provide all the required behaviour and that prevent any of the prohibited behaviour of the models being merged. In other words, merging attempts to build a new model that represents the intersection of the sets of implementations described by the models being merged.

In this thesis we concentrate on MTSs, a formalism with characteristics that make it an attractive option to be adopted by the industry when undertaking an MDSE approach. As already discussed, being a partial behaviour modeling formalism, MTSs are suitable to be used as part of an iterative and incremental software development process. Furthermore, the existence of several synthesis techniques for various specification language styles, such as Message Sequence Charts and Sequence Diagrams, Use Cases and Goal Models, which have been developed for MTSs [84, 89], represents a significant advantage when considering the potential uptake of these models as part of an MDSE approach. Moreover, as MTSs are an extension of LTSs, they benefit from existing code generating engines [57] which provide a key toolset for these models to be successfully adopted in practice by the software industry.

Although MTSs have been studied extensively, and a number of theoretical results and practical algorithms to support reasoning and elaboration of partial behaviour models expressed in this formalism have been published [51, 66, 62, 65, 33, 88, 89], there still exist several key open questions that have impacted the potential use of MTSs by software engineers. In particular, the problem of developing a complete and correct algorithm for merging MTSs has not been solved. Moreover, existing MTS semantics, *strong* and *weak* [62], assume that alphabets of the partial models being considered are the same. For partial models to support the elaboration of behaviour models in practice, an assumption that requires fixing the scope, i.e., the set of relevant observable actions, of all models *a priori* is too strong. In order to support widely used iterative, incremental software development practices, the semantics of partial behaviour models and the notion of refinement associated with it should allow for extending the alphabet of partial models as they are elaborated. As a starting point for this thesis we have focused on answering those open questions.

1.2 Motivating Example

In this section, we introduce a motivating example which we will also use as a running example throughout this thesis.

Consider a specification of software controlling a bank ATM. The specification may consist of a number of use cases exemplifying how the ATM is to be used and some properties it is expected to satisfy. An example use case is "when a user has successfully logged in, i.e., inserted a valid card and keyed in a valid password, the user must be offered the following choices: withdraw cash, balance slip or log out". Some ATMs might have an internal timeout, so that after a period of inactivity the system can log-out the user, or optionally ask the user if

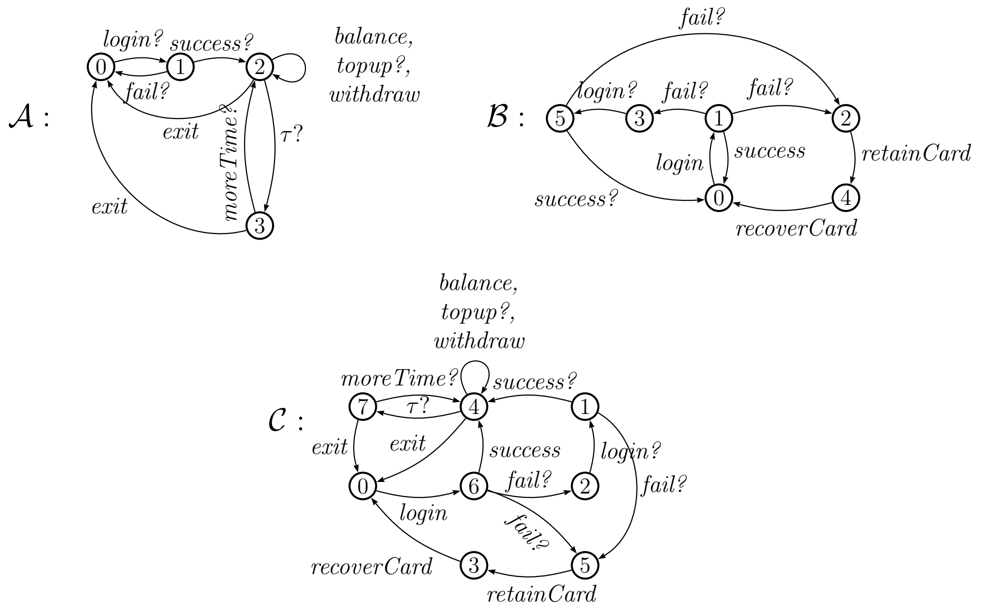


Figure 1.1: MTS and LTS models for an ATM.

more time is required. In addition, some ATMs may provide an optional feature of topping up a pay-as-you-go mobile phone. A possible safety property of an ATM is to prohibit withdrawals, balances and top-ups if the user is not logged in.

An operational model, in the form of an MTS that captures the above use case and property, is depicted in model $\mathcal{A}$ in Figure 1.1. Here, the initial state of the model is labelled 0, transitions with labels ending with a question mark represent possible but not required behaviour, while the rest of the transitions represent required behaviour. If the system has provisions for logging in the user and the login is successful, the user (in state 2) must be given a choice to withdraw cash, obtain a balance or exit. The top-up feature is optional. The time out is an internal event and therefore not visible to the user, so it is modelled with a maybe τ transition to represent that is an unobservable transition and that the system might or might not have this behaviour. No other behaviour is allowed.

Another important property of an ATM is that a user must be allowed to attempt login at least once and is not allowed to attempt to login after N failed attempts. Model $\mathcal{B}$ in Figure 1.1 depicts an MTS with $N = 2$. Note that this property does not prescribe the exact number of failed attempts after which the ATM must retain the card; hence, model $\mathcal{B}$ allows a card to be retained after one or two failed logins but forbids a third login attempt by retaining the bank card. For the user to attempt a login once more, she must recover her card from her bank branch.

ATM models do not have to be manually produced by an engineer. It might be more desirable to generate them automatically from specifications expressed in message sequence charts [54], use case diagrams [56] and structured declarative specifications such as [26]. MTS synthesis techniques have been studied [88, 89] but are beyond the scope of this work. The advantage of a synthesis approach is that it allows specifying different aspects of a system using different languages which depend on the nature of properties being expressed and preferences of the modeller. In addition, each synthesized operational model can be used to validate a specific aspect of the system-to-be.

Having validated models $\mathcal{A}$ and $\mathcal{B}$, it would be desirable to compose them to understand the implications of building a system that conforms to the requirements expressed in *both* models. Model $\mathcal{C}$ in Figure 1.1 precisely captures the behaviour prescribed by these models; it merges the required and forbidden behaviour of both models. How can such a model be constructed automatically? What are its properties? How can we guarantee that it preserves the semantics of the models being composed? How to treat models with different alphabets? In this work, we answer these questions.

1.3 Contributions

In this thesis we analyse from a Software Engineering point of view Modal Transition Systems (MTSs) as a formalism to model and elaborate system behaviour in a context where initially only partial knowledge of system behaviour is available, and iteratively this knowledge is expanded.

We first analyse MTS and its Strong Semantics as a 'platform' for behavioural model elaboration and find that some key theoretical questions were still open or only partially solved. In particular, we contribute to the theory for strong semantics of MTS with a proof that strong refinement is incomplete[1]; we characterise consistency between two models under strong semantics and extend this characterisation to a set of an arbitrary number of models; we analyse the merge operator defined as the least common refinement and prove that some models might not have any minimal common refinement although they are consistent. For consistent models that have a minimal common refinement we provide a correct and complete algorithm to compute the merge. We present and discuss the benefits of model elaboration using MTS and strong semantics, and conclude that although it has a number of convenient qualities, the requirement of a fixed alphabet is limiting when using this semantics as the basis for behavioural model elaboration.

In order to overcome this limitation we present a novel semantics for MTS called weak alphabet, which is an extension of weak semantics for MTS. This new semantics enables the modeller to increase the scope of the description as new concepts are identified, by augmenting the alphabet of the models and thus lowering the level of abstraction during the elaboration process. Furthermore, we

[1]This result was also presented in [64]

extend the previously introduced consistency characterisation for weak semantics and present sufficient conditions for consistency in weak alphabet semantics. We also develop an extension of the merge algorithm for weak and weak alphabet semantics and discuss its strengths, limitations and implications. Having introduced weak alphabet semantics, we review its adequacy for model elaboration and find that even though it can support the desired engineering process, this semantics allows some counter-intuitive refinements during the elaboration process in cases where it does not adequately preserve the branching behaviour. We therefore consider the desired behaviour a refinement should display in order to capture the intuition modellers may have of conformance and present a further new semantics, branching alphabet semantics. This semantics draws from desirable characteristics of both weak alphabet and strong semantics, i.e. it allows for extending the alphabet while preserving the branching structure. We extend the characterisation of consistency and the merge algorithm to apply to branching alphabet semantics, and prove that this semantics presents a series of desirable properties that do not hold for weak alphabet semantics, thus supporting the argument for considering branching alphabet semantics as a sound basis for model elaboration.

In addition, we study a series of algebraic properties of merge and parallel composition and their relationship with refinement, providing results that are essential to support compositional construction of system behaviour models. Also, we develop a software tool to verify the different refinement and implementation notions analysed in this work, and to compute the merge algorithm presented for the different semantics.

Finally, we exemplify the utility of the theoretical results and algorithms presented in this thesis by applying them to support behavioural model elaboration in the context of a case study.

The findings presented in this thesis are based on and extend several published papers [34, 13, 25, 33, 24, 30, 31]. This thesis should be regarded as the definitive account of this work.

1.4 Thesis Outline

The remainder of this thesis is organized as follows. In Chapter 2, we provide the theoretical background for the rest of the work. In Chapter 3, we revisit Strong Semantics, present a characterisation for consistency and introduce a new merge algorithm. In Chapter 4, we present Weak Alphabet Semantics and provide an analysis of its adequacy for model elaboration. In Chapter 5, we introduce and validate Branching Alphabet Semantics. Chapter 6 presents algebraic properties of merge. Chapter 7 provides a brief discussion on the tool we have developed for computing MTS refinement and merge, while Chapter 8 presents a case study illustrating our results and the use of the tool. Finally, in Chapter 9 we present our conclusions.

Chapter 2

Background

In this chapter we present the theoretical background for this thesis. In particular, we recall definitions and fix notation for Labelled Transition Systems (LTS), related equivalences and standard refinements, and Modal Transition Systems (MTS).

2.1 Labelled Transition Systems

Labelled Transitions Systems provide a basis for modelling and analysing system behaviour at the software architecture level. An LTS describes how a system component interacts with its environment through shared events. LTSs are equipped with a distinguished event, τ, that models any internal computation that is not observable by the environment.

Definition 2.1.1 (Labelled Transition Systems)**.** *Let States be a universal set of states, Act be a universal set of observable action labels, and let* $Act_\tau = Act \cup \{\tau\}$.

A labelled transition system *(LTS) is a tuple $P = (S, L, \Delta, s_0)$, where $S \subseteq States$ is a finite set of states, $L \subseteq Act_\tau$ is a finite set of labels, $\Delta \subseteq (S \times L \times S)$ is a transition relation between states, and $s_0 \in S$ is the initial state. We use $\alpha P = L \setminus \{\tau\}$ to denote the communicating alphabet of P.*

Figure 2.1 shows a graphical representation of two LTSs. Given an LTS $P = (S, L, \Delta, s_0)$ we say P transitions on ℓ to P', denoted $P \xrightarrow{\ell} P'$, if $P' = (S, L, \Delta, s'_0)$ and $(s_0, \ell, s'_0) \in \Delta$. Similarly, we write $P \xrightarrow{\hat{\ell}} P'$ to denote that either $P \xrightarrow{\ell} P'$ or ($\ell = \tau$ and $P = P'$) are true. We use $P \overset{\ell}{\Longrightarrow} P'$ to denote $P(\xrightarrow{\tau})^* \xrightarrow{\ell} (\xrightarrow{\tau})^* P'$. Let $w = w_1, \ldots, w_k$ be a word over Act_τ. Then $P \xrightarrow{w} P'$ means that there exist $P_0, \ldots, P_k$ such that $P = P_0$, $P' = P_k$, and $P_i \xrightarrow{w_{i+1}} P_{i+1}$ for $0 \leq i < k$. We write $P \xrightarrow{w}$ to mean $\exists P' \cdot P \xrightarrow{w} P'$. Finally, we extend $\Longrightarrow$ to words in the same way as we did for $\longrightarrow$.

2.1.1 Equivalences

Consider that $\wp$ is the universe of all LTSs.

Definition 2.1.2 (Strong Bisimulation Equivalence [77, 70])**.** *A* strong bisimulation relation *R is a binary relation on $\wp$ such that if $(P, Q) \in R$ then:*

1. $(\forall \ell, P')(P \xrightarrow{\ell} P') \implies (\exists Q' \cdot Q \xrightarrow{\ell} Q' \ \wedge \ (P', Q') \in R)$
2. $(\forall \ell, Q')(Q \xrightarrow{\ell} Q') \implies (\exists P' \cdot P \xrightarrow{\ell} P' \ \wedge \ (P', Q') \in R)$

Two LTSs P and Q are strong equivalent*, written $P \sim Q$, if $\alpha P = \alpha Q$ and there exists a strong bisimulation relation R such that $(P, Q) \in R$.*

This notion of Bisimulation Equivalence was originally presented by Park [77] in 1981 and used by Milner in [43, 70] and in a different formulation was already presented by Milner [42] in 1980. Informally, two models are strong equivalent if their initial states are strong equivalent, and two states are strong equivalent if, whenever one action can be executed in one of them leading to a state B, the other can execute the same action reaching a state B', where B' is again strong equivalent to B. This equivalence does not distinguish τ transitions as special or unobservable actions. A property of this equivalence is that it respects the branching structure of processes [92].

Consider the LTSs shown in Figure 2.1. These two models are an example of strong equivalent models, and the bisimulation relation between them is

$$R = \{(a_0, b_0), (a_1, b_1), (a_2, b_2), (a_0, b_3), (a_1, b_4), (a_2, b_5)\}.$$

On the other hand, Figure 2.2 shows an example of two LTSs that are not strong equivalent. There does not exist a strong bisimulation relation for these models because state 1 of the model in Figure 2.2(a) cannot be related to any state of the model in Figure 2.2(b).

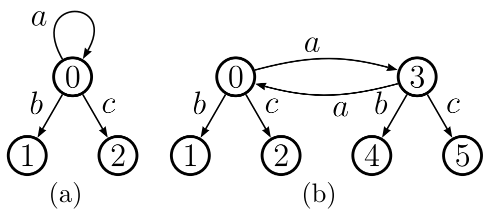


Figure 2.1: Example of strong equivalent models.

Definition 2.1.3 (Weak Bisimulation Equivalence [72]). *A* weak bisimulation relation *R is a binary relation on $\wp$ such that if $(P, Q) \in R$ then:*

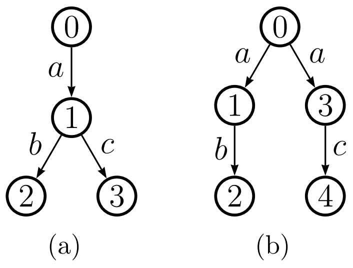


Figure 2.2: Example of not strong equivalent models.

1. $(\forall \ell, P')(P \xrightarrow{\ell} P') \implies (\exists Q' \cdot Q \overset{\hat{\ell}}{\Longrightarrow} Q' \;\wedge\; (P', Q') \in R)$
2. $(\forall \ell, Q')(Q \xrightarrow{\ell} Q') \implies (\exists P' \cdot P \overset{\hat{\ell}}{\Longrightarrow} P' \;\wedge\; (P', Q') \in R)$

Two LTSs P and Q are weak equivalent*, written $P \approx_{\mathrm{w}} Q$, if $\alpha P = \alpha Q$ and there exists a weak bisimulation relation R such that $(P, Q) \in R$.*

This equivalence compares the observational behaviour of models ignoring silent actions (τ-transitions). Some authors call this equivalence *observational equivalence*, but we are going to use this expression to refer to any equivalence that considers τ-transitions as silent actions. Weak bisimulation equivalence is coarser than strong equivalence and does not preserve the branching structure of processes as it is shown in [91].

Figure 2.3 shows an example of two models that are not strong equivalent but weak equivalent. The weak bisimulation relation between them is

$$R = \{(a_0, b_0), (a_1, b_1), (a_2, b_1), (a_3, b_3)\}.$$

Definition 2.1.4 (Branching Bisimulation Equivalence [93])**.** *A* branching bisimulation relation *R is a binary relation on $\wp$ such that if $(P, Q) \in R$ then:*

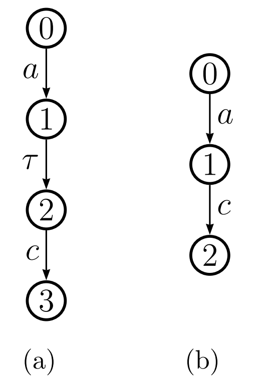


Figure 2.3: Example of weak equivalent models that are not strong equivalent.

1. $(\forall \ell, P')(P \xrightarrow{\ell} P') \implies (\exists Q', Q'' \cdot Q \overset{\hat{\tau}}{\Longrightarrow} Q' \xrightarrow{\hat{\ell}} Q'' \;\wedge\; (P, Q'), (P', Q'') \in R)$
2. $(\forall \ell, Q')(Q \xrightarrow{\ell} Q') \implies (\exists P', P'' \cdot P \overset{\hat{\tau}}{\Longrightarrow} P' \xrightarrow{\hat{\ell}} P'' \;\wedge\; (P', Q), (P'', Q') \in R)$

Two LTSs P and Q are branching equivalent*, written $P \approx_{\mathrm{b}} Q$, if $\alpha P = \alpha Q$ and there exists a branching bisimulation relation R such that $(P, Q) \in R$.*

Lemma 2.1.5 (Stuttering Lemma [93]). *Let R be the largest branching bisimulation between P and Q and $(r, s) \in R$. If $r \xrightarrow{\tau} r_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} r_m \xrightarrow{\tau} r' (m \geq 0)$ is a path such that $(r', s) \in R$ then $(r_i, s) \in R \;\forall\; i \leq m$.*

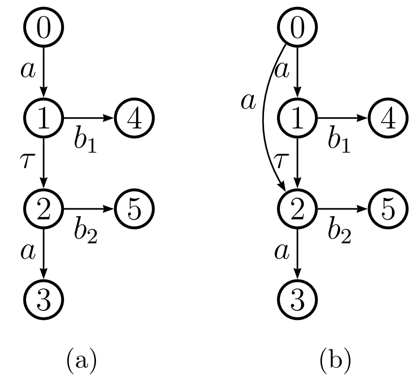


Figure 2.4: Example of weak but not branching equivalent models.

Branching bisimulation equivalence is an observational equivalence coarser than strong equivalence and finer than weak bisimulation equivalence. This equivalence is the coarsest equivalence that preserves the branching structure of processes [92]. The Stuttering Lemma will have an important role in the results we present in section 5.2.2.

Figure 2.4 shows an example of two models that are weak equivalent but not branching equivalent. If we make a comparative analysis of models (a) and (b) we can see that while from the initial state in model (a) we can take transition a in only one way, in model (b) there are two different possibilities to take this transition from state 0. One of those two possibilities leads to a state from where both b_1 and b_2 transitions can be taken. This is the same that happens if we take transition a in model (a). However, if from the initial state in model (b) we take transition a leading to state 2 then the possibility of taking b_1 is discarded before having the chance of taking it, which is never the case in model (a). Therefore, we can conclude that these two models do not have the same branching structure.

2.1.2 Refinements

In the previous section we presented a series of equivalences over LTSs. While all these equivalences determine whether two models have or do not have the same behaviour, they differ in the criteria used to interpret the behaviour given by a model.

In the context of evolving a software model we need to be able to add further information to the model while more knowledge regarding the system is acquired. This implies that we do not only need to be able to assess if two models are equivalent or not but to define if a model with more information actually refines a previous one. In order to do so, we will need to find a suitable semantics that establishes an order between models.

In this section we present two different refinement notions over LTSs and we analyse why they do not allow us to evolve a model according to our expectations.

Definition 2.1.6 (Strong Simulation [43]). *A* strong simulation relation R *is a binary relation on* $\wp$ *such that if* $(P, Q) \in R$ *then:*

1. $(\forall \ell, P')(P \xrightarrow{\ell} P') \implies (\exists Q' \cdot Q \xrightarrow{\ell} Q' \ \wedge \ (P', Q') \in R)$

Given LTSs P *and* Q, *we say that* Q simulates P, *written* $P \sqsubseteq_{\mathrm{s}} Q$, *iff* $\alpha P = \alpha Q$ *and there exists a strong simulation relation* R *such that* $(P, Q) \in R$.

While Strong Bisimulation defines an equivalence, Strong Simulation defines a preorder over LTSs. Note that $P \sim Q \quad \implies \quad P \sqsubseteq_{\mathrm{s}} Q \ \wedge \ Q \sqsubseteq_{\mathrm{s}} P$. A classic example that ilustrates that the reciprocal of this property does not hold is presented in [16].

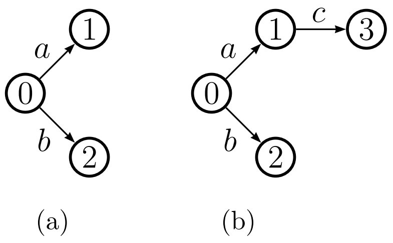


Figure 2.5: Example of LTS refinement by simulation.

A standard refinement notion between LTSs considers that P refines Q if P simulates Q. i.e. $Q \sqsubseteq_{\mathrm{s}} P$. One of the properties that characterize this refinement is that it reduces the degree of non-determinism. With this refinement we know that any behaviour of Q is a valid behaviour of P. However, using this semantics it is not possible to ensure that a forbidden behaviour for the system which is captured in Q will be preserved in the refined model P. For example, the model shown in figure 2.5(b) refines the one shown in figure 2.5(a) and it can be easily seen that model 2.5(b) has the possibility of taking transition c after taking a in spite of this being forbidden in model 2.5(a). Moreover, a model consisting of a state with a self-transition for every element of the alphabet is a refinement of any other model under this semantics.

On the other hand, if we consider that P refines Q if Q simulates P, i.e. $P \sqsubseteq_{\mathrm{s}} Q$, we know that any behaviour of P is a valid behaviour of Q, but not the other way around. Therefore, if Q determines that certain behaviour is forbidden in a system we know that it will also be forbidden by P since any behaviour in P can be simulated by Q. However, using this semantics it is not possible to ensure that a desired behaviour for the system which is depicted in Q will be preserved in the refined model P. For example, the model shown in figure 2.5(a) refines the one shown in figure 2.5(b) and it can be easily seen that model 2.5(a) loses the possibility of taking transition c after taking transition a. Moreover, a model consisting of a state with no transitions is, considered by this semantics, a refinement of any other model.

Definition 2.1.7 (Traces). *A trace is a sequence (finite or infinite) $w \in Act^*$. Given an LTS P, the set of* traces *of P is defined as follows:*

$$TRACES(P) = \{\ w \in Act^* \mid P \xrightarrow{w}\ \}$$

Definition 2.1.8 (Trace Refinement [4]). *Let P and Q be LTSs. We say that Q is a* trace refinement *of P, written $P \sqsubseteq_{\mathrm{tr}} Q$, iff $TRACES(P) \supseteq TRACES(Q)$.*

Suppose an LTS model P describes the knowledge we have at a certain stage of the development process. Then, according to trace refinement semantics, an LTS model Q will be a valid implementation of P iff the set of traces of Q is a subset of the traces of P. Therefore, P determines the set of all possible traces for the system but cannot guarantee any of them. This means that while P describes the "maximum" behaviour possible for the implementations it cannot assure any required behaviour will be preserved. For example, according to this semantics, a model consisting of only one state and no transitions is a possible implementation of any system. Moreover, the definition of trace refinement is such that the inclusion of any trace is independent from the inclusion of other traces (except for its prefixes). Consequently, using this semantics it is not possible to describe a requirement such as 'if the implementation has this specific behaviour then it must have this other behaviour'.

Bearing in mind our aim of supporting the evolution of behavioural models having partial information as a starting point and enriching the model while more requirements are gathered, the refinements presented in this section have the following limitation: they consider the less refined model to completely describe either the maximum or the minimum allowed behaviour for the system. In the first case all possible behaviour is specified whereas in the second case the model describes all the required behaviour for the system. This means only one bound is specified, i.e. either the lower or upper bound, and hence the other bound remains open. This limitation is not due to the analysed refinement notions in themselves, failures refinement [82] and testing refinement [18], for example, have similar problems. In fact, this limitation is an intrinsic characteristic of LTS models. Specifically, using these models it is not possible to identify which aspects of the system have already been defined and which still have to be refined. This is true both for the expected and the forbidden behaviour. For this reason we will consider MTSs since they allow us to explicitly specify which behaviour is required, possible or forbidden.

2.2 Modal Transition Systems

Definition 2.2.1 (Modal Transition Systems [66]). *A modal transition system (MTS) M is a structure $(S, L, \Delta^r, \Delta^p, s_0)$, where $\Delta^r \subseteq \Delta^p$, (S, L, Δ^r, s_0) is an LTS representing required transitions of the system and (S, L, Δ^p, s_0) is an LTS representing possible (but not necessarily required) transitions of the system. We use $\alpha M = L \setminus \{\tau\}$ to denote the communicating alphabet of M.*

Figure 2.6 shows a graphical representation of an MTS. Transition labels that have a question mark are those in $\Delta^p - \Delta^r$. We refer to these transitions as "maybe" transitions, to distinguish them from required ones (those in Δ^r). In this example, the transition from state 1 to 3 is a maybe transition by c, while the other two transitions are required. The labels of the states have no meaning and are used for reference only. In the rest of this thesis we will denote the initial state with the label 0. Given an MTS $M = (S, L, \Delta^r, \Delta^p, s_0)$ we say M transitions on ℓ through a required transition to M', denoted $M \xrightarrow{\ell}_{\mathrm{r}} M'$, if $M' = (S, L, \Delta^r, \Delta^p, s'_0)$ and $(s_0, \ell, s'_0) \in \Delta^r$, and M transitions through a possible transition, denoted $M \xrightarrow{\ell}_{\mathrm{p}} M'$, if $(s_0, \ell, s'_0) \in \Delta^p$. Similarly, for $\gamma \in \{r, p\}$ we write $M \xrightarrow{\hat{\ell}}_\gamma M'$ to denote that either $M \xrightarrow{\ell}_\gamma M'$ or ($\ell = \tau$ and $P = P'$) are true, and we use $P \xRightarrow{\ell}_\gamma P'$ to denote $P(\xrightarrow{\tau}_\gamma)^* \xrightarrow{\ell}_\gamma (\xrightarrow{\tau}_\gamma)^* P'$.

Note that LTSs are a special type of MTSs that do not have maybe transitions, i.e. $\Delta^r = \Delta^p$

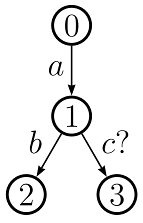


Figure 2.6: Example of an MTS.

2.2.1 Notation

Given an MTS $M = (S, A, \Delta^r, \Delta^p, s_0)$ and an action $\ell \in \mathit{Act}$, we say that:

- M has a *required* transition on ℓ (denoted $M \xrightarrow{\ell}_{\mathrm{r}} M'$) iff $(s_0, \ell, s'_0) \in \Delta^r$ and $M' = (S, A, \Delta^r, \Delta^p, s'_0)$.
- M has a *possible* transition on ℓ (denoted $M \xrightarrow{\ell}_{\mathrm{p}} M'$) iff $(s_0, \ell, s'_0) \in \Delta^p$ and $M' = (S, A, \Delta^r, \Delta^p, s'_0)$.
- M has a *maybe* transition on ℓ (denoted $M \xrightarrow{\ell}_{\mathrm{m}} M'$) iff $(s_0, \ell, s'_0) \in \Delta^p$, $(s_0, \ell, s'_0) \notin \Delta^r$ and $M' = (S, A, \Delta^r, \Delta^p, s'_0)$.
- We write $M \xrightarrow{\ell}_\gamma$ to mean $\exists M' \cdot M \xrightarrow{\ell}_\gamma M'$, where $\gamma \in \{\mathrm{r}, \mathrm{p}\}$.
- M *prohibits* ℓ (denoted $M \overset{\ell}{\not\longrightarrow}$) iff M does not have a possible transition on ℓ, i.e., $\forall s'_0 \in S \cdot (s_0, \ell, s'_0) \notin \Delta^p$.

Let $w = l_1, \ldots, l_k$ be a word over Act_τ^*. We use the following notation assuming $\ell \in Act_\tau$:

- For $\gamma \in \{r, p\}$, $M \xrightarrow{w}_\gamma M'$ denotes $M \xrightarrow{l_1}_\gamma \ldots \xrightarrow{l_k}_\gamma M'$.
- For $\gamma \in \{r, p\}$, $M \xrightarrow{\hat{\ell}}_\gamma M'$ denotes either that $M \xrightarrow{\ell}_\gamma M'$ or that $M = M'$ and $\ell = \tau$.
- For $\gamma \in \{r, p\}$, $M \xRightarrow{\ell}_\gamma M'$ denotes $M(\xrightarrow{\tau}_\gamma)^*(\xrightarrow{\ell}_\gamma)(\xrightarrow{\tau}_\gamma)^* M'$. Similarly, $M \xRightarrow{\hat{\ell}}_\gamma M'$ denotes $M(\xrightarrow{\tau}_\gamma)^*(\xrightarrow{\hat{\ell}}_\gamma)(\xrightarrow{\tau}_\gamma)^* M'$.
- For $\gamma \in \{r, p\}$, we extend $\Longrightarrow_\gamma$ to words the same way as we do $\longrightarrow_\gamma$.
- For $\gamma \in \{r, p\}$, we write $s \xrightarrow{\ell}_\gamma s'$ to denote $M_s \xrightarrow{\ell}_\gamma M_{s'}$ (and similarly, for $\Longrightarrow_\gamma$).

Chapter 3

Revisiting Strong Semantics

3.1 Definition

When considering LTS, *strong semantics* refers to the semantics given by strong bisimulation. One of the particularities of this semantics is that it lacks a notion of unobservable or internal action, i.e. τ-labelled transitions. Larsen has extended this semantics over MTS [66].

Strong refinement of MTS captures the notion of elaboration of a partial description into a more comprehensive one, in which some knowledge over the maybe behaviour has been gained. It can be seen as being a "more defined than" relation between two partial models. Intuitively, refinement in MTS is about converting maybe transitions into required transitions or removing them altogether: an MTS N refines M if N preserves all of the required and all of the proscribed behaviours of M. Alternatively, an MTS N refines M if N can simulate the required behaviour of M, and M can simulate the possible behaviour of N.

Consider that δ is the universe of all MTSs.

Definition 3.1.1 (Strong Refinement)**.** *A* strong refinement relation *R is a binary relation on δ such that if $(M, N) \in R$ then:*

1. $(\forall \ell, M')(M \xrightarrow{\ell}_{\mathrm{r}} M') \implies (\exists N' \cdot N \xrightarrow{\ell}_{\mathrm{r}} N' \ \wedge \ (M', N') \in R)$
2. $(\forall \ell, N')(N \xrightarrow{\ell}_{\mathrm{p}} N') \implies (\exists M' \cdot M \xrightarrow{\ell}_{\mathrm{p}} M' \ \wedge \ (M', N') \in R)$

Given MTSs M and N, we say that N is a strong refinement *of M, written $M \preceq N$, iff $\alpha M = \alpha N$ and there exists a strong refinement relation R such that $(M, N) \in R$.*

Note that the second condition guarantees that if N has a required transition, M has a maybe or a required transition, whereas if N has a maybe transition, then M has a maybe transition – otherwise, the first condition is violated. It is also interesting to note how similar this definition is to that of strong bisimulation in Definition 2.1.2.

Consider model $\mathcal{A}$ shown in Figure 1.1. If modellers decide not to support topping up mobile phones, and also for security reasons not to offer the user the option to request more time once the inactivity time out triggers, then the model that would represent these decisions is the one shown in Figure 3.1. According to strong semantics this latter model is a valid possible evolution of model $\mathcal{A}$ since it is a valid refinement, that incorporates as new knowledge that the *topup* and *moreTime* options have been removed from the functionalities of the system. The refinement relation between these models is

$$R = \{(0,0), (1,1), (2,2), (3,3)\}.$$

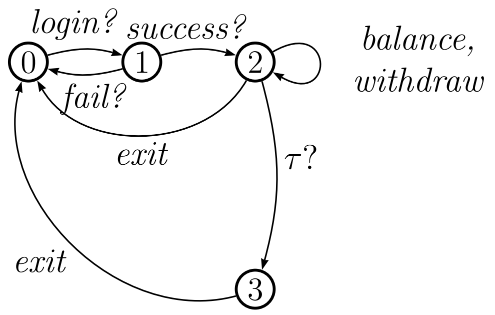


Figure 3.1: A possible evolution of model $\mathcal{A}$ where the ATM does not support topping up mobile phones, and it does not give the user the option to request more time once the inactivity time out is triggered.

If an MTS I has no maybe transitions and is a valid refinement of an MTS M, I can be considered as an LTS and in this case we say that I is an implementation of model M.

Definition 3.1.2 ((Strong) Implementation)**.** *We say that an LTS $I = (S_I, L_I, \Delta_I, i_0)$ is a* (strong) implementation *of an MTS $M = (S_M, L_M, \Delta^r_M, \Delta^p_M, m_0)$, written $M \preceq I$, if $M \preceq M_I$ with $M_I = (S_I, L_I, \Delta_I, \Delta_I, i_0)$. We also define the set of implementations of M as $\mathcal{I}[M] = \{I \in \wp \mid M \preceq I\}$.*

Property 3.1.3 (Soundness)**.** *Given MTSs M and N, if $M \preceq N$ then $\mathcal{I}[M] \supseteq \mathcal{I}[N]$.*

Property 3.1.3 is known as the soundness property. We say that strong refinement is sound with respect to implementation inclusion. Considering that strong refinement is transitive [66] it is straightforward to prove Property 3.1.3. Hence when a model is evolved into a more refined one no new possible implementations for the system will be added. In fact, when a model is enriched with more requirements the set of possible implementations can only be reduced.

3.2 Completeness

A question that had been open is whether the reciprocal of Property 3.1.3 is also valid. In other words, whether inclusion of implementations implies refinement, i.e. $\mathcal{I}[M] \supseteq \mathcal{I}[N] \implies M \preceq N$. If this is the case, we say that the refinement is complete.

In this section we will present a counter example of completeness and discuss the idea of defining refinement in terms of implementations. The following counter example was developed simultaneously by the author of this thesis and other authors, and was presented in [32, 64, 28].

Consider models $\mathtt{A}$ and $\mathtt{B}$ depicted in Figure 3.2. It is easy to see that models $\mathtt{I}_1$, $\mathtt{I}_2$ and $\mathtt{I}_3$ on the same figure represent all the different implementations, up to equivalence, of $\mathtt{A}$ and $\mathtt{B}$, therefore $\mathcal{I}[\mathtt{A}] = \mathcal{I}[\mathtt{B}]$. However, $\mathtt{B} \not\preceq \mathtt{A}$ since it is impossible to construct a refinement relation between $\mathtt{B}$ and $\mathtt{A}$ because state 1 of model $\mathtt{B}$ cannot be related by a strong refinement relation with any state of $\mathtt{A}$. Therefore, these models represent a counter example of completeness for strong refinement as they have the same set of implementations but $\mathtt{B} \not\preceq \mathtt{A}$.

Refinements between MTSs can be defined giving an operational definition as in Definition 3.1.1 [66]. However, the previous counter example showed that strong refinement as defined in Definition 3.1.1 is not complete. Alternatively, we can define refinements between MTSs using a declarative definition based on inclusion of implementations. In this case, the refinement is complete by definition. In order to clarify which of these notions we are referring to, when necessary we will adopt the terminology introduced on [64], which names *modal refinement* to the refinement given by an operational refinement relation as in Definition 3.1.1, and

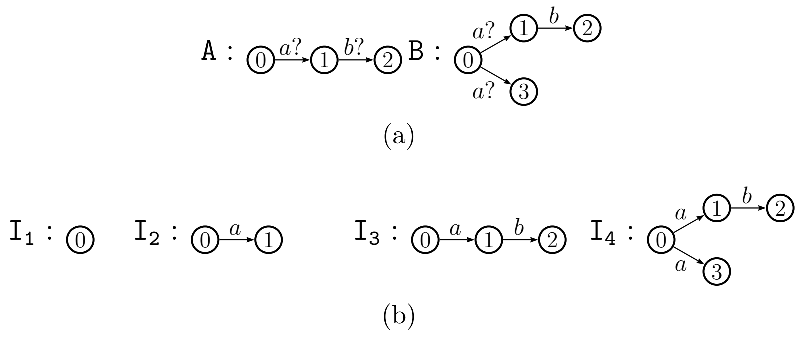


Figure 3.2: Counter example of completeness for strong refinement. a) $\mathcal{I}[\mathtt{A}] = \mathcal{I}[\mathtt{B}]$ but $\mathtt{B} \not\preceq \mathtt{A}$ b) Implementations of A and B.

thorough refinement to the refinement given by inclusion of implementations. For the remainder of this thesis, when we just say *refinement* we will be referring to *modal refinement*.

We now provide a formal definition of *strong thorough refinement*.

Definition 3.2.1 (Strong Thorough Refinement)**.** *An MTS N is a* strong thorough refinement *of an MTS M, written $M \preceq_{\mathrm{t}} N$, iff $\mathcal{I}[N] \subseteq \mathcal{I}[M]$.*

It can be noted that *strong thorough refinement* is, by definition, sound and complete. However, checking thorough refinement is EXPTIME-complete [3, 7]. Therefore, although incomplete, modal refinement is a more appealing and useful notion of refinement from an engineering point of view since it can be computed in polynomial time [34], which allows practitioners to build tools for checking refinement and operate between models.

3.3 Consistency

Intuitively two models that provide partial descriptions of the same system are consistent if the required and forbidden behaviour described by each of them are compatible. Some of the maybe behaviour of one may be required or forbidden in the other but required behaviour in one of the models cannot be forbidden in the other. Formally, consistency is defined as the existence of a common implementation as stated in Definition 3.3.1.

Definition 3.3.1. (Consistency) *Two MTSs M and N are consistent if there exists an LTS I such that I is a common implementation of M and N.*

Consider model $\mathcal{H}$ in Figure 3.3 that specifies an ATM in which, in addition to the top-up feature being enabled, a withdrawal automatically logs the user out (to prevent the user from forgetting her card). This model is inconsistent with previous ATM models such as $\mathcal{A}$ in Figure 1.1 which forbids logging in until an exit action has occurred. It is therefore impossible to build an ATM that satisfies both model $\mathcal{H}$ and model $\mathcal{A}$.

Checking if two models are consistent is of clear use to engineers that have multiple partial descriptions of system behaviour. In particular, consistency is a pre-condition for merging models as there can be no most abstract common refinement if there are no common refinements.

However, this problem has not been solved for strong semantics, i.e. given two MTSs determine if there exists a model which is a refinement of both models using strong refinement. In [65] the *independence* relation was introduced, which provides an approximation to consistency but it does not characterise it.

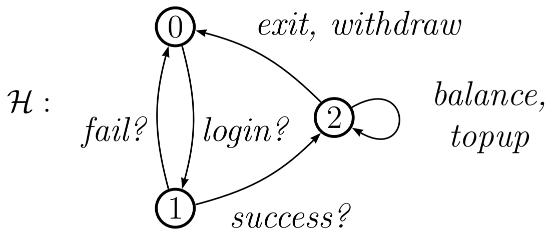


Figure 3.3: Example of an inconsistent MTS. Model $\mathcal{H}$ is inconsistent with model $\mathcal{A}$.

Definition 3.3.2. (Independence [65]) *An* independence relation *R is a binary relation on δ such that if $(S,T) \in R$ then:*

1. $(\forall \ell, S')(S \xrightarrow{\ell}_{\mathrm{r}} S' \implies (\exists! T')(T \xrightarrow{\ell}_{\mathrm{p}} T' \wedge (S',T') \in R))$
2. $(\forall \ell, T')(T \xrightarrow{\ell}_{\mathrm{r}} T' \implies (\exists! S')(S \xrightarrow{\ell}_{\mathrm{p}} S' \wedge (S',T') \in R))$
3. $(\forall \ell, S', T')(S \xrightarrow{\ell}_{\mathrm{p}} S' \wedge T \xrightarrow{\ell}_{\mathrm{p}} T') \implies (S',T') \in R$

Two models M and N are independent if there exists an independence relation R such as $(M,N) \in R$.

Intuitively, the *independence* notion was defined to capture when two models are not contradictory and is used to define merge (called conjunction in [65]) on MTSs. However, in the following example we show that independence does not characterise when two models are consistent.

Consider models $\mathcal{A}$ and $\mathcal{B}$ of Figure 3.4 which are not independent: if $(0,0)$ were in an independence relation then $(1,1)$ would have to be as well because of rule 3. However, rule 1 would be violated because model $\mathcal{B}$ can do a required action on b from state 1 but model $\mathcal{A}$ cannot follow this action with a possible b. Therefore $(1,1)$ cannot be in the relation, which implies that $(0,0)$ cannot be in it either. Although these models are not independent they are consistent since they have model $\mathcal{C}$ as a common refinement.

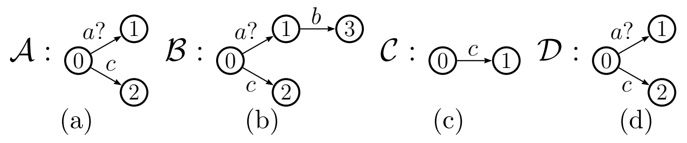


Figure 3.4: Example of consistent but not independent models.

We now present a new relation, *strong consistency relation*, and show that it characterises consistency, i.e. we prove that two models are consistent if and only if they can be related by the strong consistency relation.

Definition 3.3.3. (Strong Consistency Relation) *A strong consistency relation C is a binary relation on δ such that if $(M, N) \in C$ then:*

1. $(\forall \ell, M')(M \xrightarrow{\ell}_{\mathrm{r}} M') \implies (\exists N')(N \xrightarrow{\ell}_{\mathrm{p}} N' \wedge (M', N') \in C)$
2. $(\forall \ell, N')(N \xrightarrow{\ell}_{\mathrm{r}} N') \implies (\exists M')(M \xrightarrow{\ell}_{\mathrm{p}} M' \wedge (M', N') \in C)$

Intuitively, this relation requires that one model provides as possible behaviour at least all the required behaviour of the other, and vice versa. For example,

$$R = \{(0, 0), (2, 2)\}$$

is a strong consistency relation between models $\mathcal{A}$ and $\mathcal{B}$ in Figure 3.4. We can see that the strong consistency relation is weaker than the independence relation since it does not have condition 3 of the independence relation and conditions 1 and 2, although similar to those of the independence relation, do not require a deterministic choice. This weaker relation relates models which were related by independence but also relates pairs of models that are not independent, such as $\mathcal{A}$ and $\mathcal{B}$. Furthermore the strong consistency relation only relates those models which are consistent.

Theorem 3.3.4. (Strong Consistency Relation Characterizes Consistency) *Two MTSs M and N are consistent if and only if there exists a strong consistency relation C_{MN} such that (M, N) is contained in C_{MN}.*

Proof. $\Leftarrow$) Let CI be a LTS defined by

$$CI = (C_{MN}, Act, \Delta_{CI}, (M_0, N_0))$$

where Δ_{CI} is the smallest relation that satisfies the following rules, assuming that $\{(M, N), (M', N') \subseteq C_{MN}\}$.

$$\text{RP}\ \frac{M \xrightarrow{\ell}_{\text{r}} M', N \xrightarrow{\ell}_{\text{p}} N'}{(M,N) \xrightarrow{\ell} (M',N')} \qquad \text{PR}\ \frac{M \xrightarrow{\ell}_{\text{p}} M', N \xrightarrow{\ell}_{\text{r}} N'}{(M,N) \xrightarrow{\ell} (M',N')}$$

It is easy to prove that $M \preceq CI$ using that

$$R = \{(M, (M, N)) \mid (M, N) \in C_{MN}\}$$

is an implementation relation between M and CI.

$\Rightarrow$) Since M and N are consistent we can take an LTS CI such that $M \preceq CI$ and $N \preceq CI$. By definition of strong semantics there exist R_M and R_N implementation relations between M and CI, and between N and CI respectively.

Let C_{MN} be a relation defined by $C_{MN} = R_M \circ R_N^{-1}$. It can easily be proven that C_{MN} is a strong consistency relation between M and N. $\square$

Note that the Strong Consistency Relation is equivalent to bisimulation when restricted to LTSs (i.e. MTS with identical sets of required and possible transitions). This result is as expected considering that an LTS is an MTS which characterises only one implementation, itself. Hence, an LTS can only be consistent with any LTS that is equivalent to it; equivalence which in this case is that of LTS bisimulation.

Another observation worth mentioning is that the proof of Theorem 3.3.4 only uses the implementation notion, thus the strong consistency characterization is the same for strong modal refinement and strong thorough refinement. This result is also interesting because it shows that even though checking thorough refinement is EXPTIME-complete [7], checking if two models are consistent under that semantics is polynomial on the size of the models.

We have developed a fixed point algorithm for checking consistency that starts with the Cartesian product of the states and iteratively eliminates the pairs that are not valid according to the strong consistency relation. The completeness and correctness proofs for this algorithm are straightforward and are provided for a similar algorithm in Section 4.2.2. This algorithm is polynomial and its time complexity is upper bounded by $O(m.n^4.log(n))$ while the space complexity is $O(n^2)$, where n and m are the maximum amount of states and transitions respectively between both models.

3.4 Merging MTS

In this section, we solve the problem of merging MTS under strong semantics. We first recall the definition of MTS merging [65, 90], then analyse the limitations of existing algorithms, and finally present a merge algorithm that is complete and correct for MTSs with identical alphabets.

The intuition captured by merge is that of augmenting the knowledge we have of the behaviour of a system by combining what we know from two partial descriptions of the system. The notion of refinement underlies this intuition as it captures the "more defined than" relation between two partial models.

Definition 3.4.1. (Common Refinement) *Given a refinement notion, $\preceq$, we say that a modal transition system P is a* common refinement *(CR) of modal transition systems M and N iff $M \preceq P$ and $N \preceq P$.*

We write $\mathcal{CR}(M, N)$ to denote the set of common refinements of models M and N.

A common refinement cannot leave as undefined behaviour that is already defined in M or N. Although a common refinement P of M and N preserves the required and proscribed behaviour of M and N, it may be too refined. The result of merging should not only preserve required and proscribed behaviour of M and N but also introduce as few decisions on maybe behaviour as possible. In other words, the merge of M and N should characterise all LTS that are implementations of both M and N. Indeed, this corresponds to the least (with respect to refinement) common refinement of M and N.

Definition 3.4.2. (Least Common Refinement) *A modal transition system P is the least common refinement (LCR) of modal transition systems M and N if P is a common refinement of M and N, and for any common refinement Q of M and N, $P \preceq Q$.*

An LCR of the original systems may not exist for two reasons. First, it is possible that no common refinement exists as the models might not be consistent. Second, a common refinement may exist, but there may be no least one. For example, models $\mathcal{O}$ and $\mathcal{P}$ in Figure 3.5 are consistent and have models $\mathcal{Q}$ and $\mathcal{R}$ as common refinements. These common refinements are not comparable (neither is a refinement of the other) and it is not possible to find common refinements of $\mathcal{O}$ and $\mathcal{P}$ which are less refined than $\mathcal{Q}$ or $\mathcal{R}$. Hence, we refer to $\mathcal{Q}$ and $\mathcal{R}$ as the minimal common refinements of $\mathcal{O}$ and $\mathcal{P}$. We now provide a formal definition of minimal common refinement.

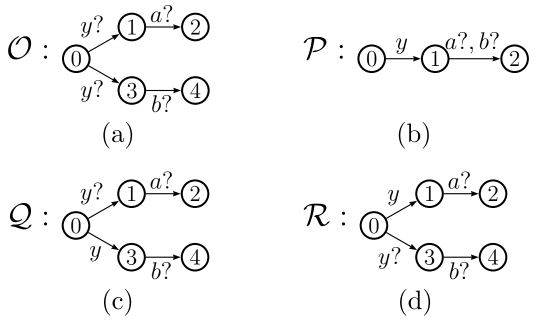


Figure 3.5: Example of two consistent models that do not have an LCR but two non-equivalent minimal common refinements.

Definition 3.4.3. (Minimal Common Refinement) *Given a refinement notion, $\preceq$, an MTS P is a* minimal common refinement *(MCR) of MTSs M and N if $P \in \mathcal{CR}(M, N)$, and for all $Q \in \mathcal{CR}(M, N)$ if $Q \preceq P$, then $P \preceq Q$.*

We write $\mathcal{MCR}(M, N)$ to denote the set of MCRs of models M and N.

Figure 3.6 provides an abstract summary of the concepts discussed in this section. In this figure, nodes depict models and arrows depict refinements (i.e., an edge from P to Q indicates that P is refined by Q). For simplicity, we do not depict refinements that can be inferred by transitive closure of the ones depicted.

In conclusion, what should be the result of merging two consistent modal transition systems, M and N? If $\mathcal{LCR}_{M,N}$ exists, then this is the desired result of the

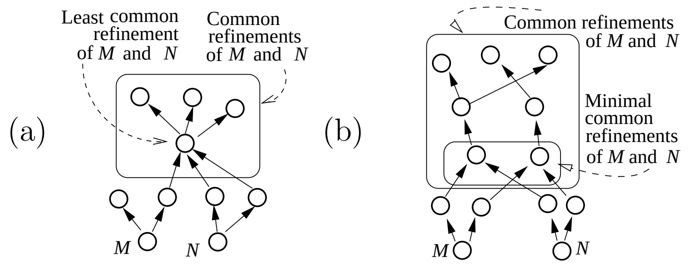

(a)
Least common refinement of M and N
Common refinements of M and N
M
N
(b)
Common refinements of M and N
Minimal common refinements of M and N
M
N

Figure 3.6: Common refinements for consistent models M and N: (a) M and N have a least common refinement; (b) M and N have no least common refinement.

merge. However, if M and N are consistent but their LCR does not exist, then the merge process should result in one of the MCRs of M and N. Model merging should support the modeller in choosing the most appropriate MCR.

3.4.1 Limitations of Existing Algorithms

In [65] an operation between two models called conjunction is defined. This operation when applied to independent models gives their LCR.

Definition 3.4.4. (Conjunction) [65] *Let M and N be MTSs, the conjunction of M and N is defined as $M \wedge N = (S_M \times S_N, L, \Delta^r_{M\wedge N}, \Delta^p_{M\wedge N}, (m_0, n_0))$, where $\Delta^r_{M\wedge N}, \Delta^p_{M\wedge N}$ are the smallest relations which satisfy the following rules:*

$$RP\ \frac{M\xrightarrow{\ell}_{\mathrm{r}}M',\ N\xrightarrow{\ell}_{\mathrm{p}}N'}{(M,N)\xrightarrow{\ell}_{\mathrm{r}}(M',N')}\qquad PR\ \frac{M\xrightarrow{\ell}_{\mathrm{p}}M',\ N\xrightarrow{\ell}_{\mathrm{r}}N'}{(M,N)\xrightarrow{\ell}_{\mathrm{r}}(M',N')}\qquad PP\ \frac{M\xrightarrow{\ell}_{\mathrm{p}}M',\ N\xrightarrow{\ell}_{\mathrm{p}}N'}{(M,N)\xrightarrow{\ell}_{\mathrm{p}}(M',N')}$$

The limitation of the conjunction operator is that there are models with an LCR that the operator fails to produce. In fact, in these cases the operator does not produce a common refinement. Consider models $\mathcal{A}$ and $\mathcal{B}$ in Figure 3.4. The LCR of these models is $\mathcal{C}$. However, the conjunction as defined in [65] is model $\mathcal{D}$, which is not a common refinement of $\mathcal{A}$ and $\mathcal{B}$ ($\mathcal{C}$ and $\mathcal{D}$ are also depicted in

Figure 3.4). This problem occurs when two models are not independent but are consistent.

To overcome the problems of the conjunction operator, a variation of this operator was proposed initially in [90] and then improved in [10]. This new operator, called $+_{cr}$, generalises the problem addressed in [65] to support merging models under weak semantics. When the models being merged have no unobservable actions, the problem is that of merge under strong semantics, as discussed below.

Definition 3.4.5. (The $+_{cr}$ operator) [10] *Let M and N be MTSs and let C_{MN} be the largest strong consistency relation between them. The $+_{cr}$ operator between M and N is defined as $M +_{cr} N = (C_{MN}, L, \Delta^r_{M+_{cr}N}, \Delta^p_{M+_{cr}N}, (m_0, n_0))$, where $\Delta^r_{M+_{cr}N}, \Delta^p_{M+_{cr}N}$ are the smallest relations which satisfy rules RP, PR, PP of Def. 3.4.4.*

Note that the difference between the conjunction operator and the $+_{cr}$ operator is that the latter only considers pairs of states which are consistent. Returning to the example of Figure 3.4, the consistency relation between models $\mathcal{A}$ and $\mathcal{B}$ is $\{(0,0), (2,2)\}$, therefore if we apply the $+_{cr}$ to these models the rules that can be applied are PR and RP producing a required transition between the state $(0,0)$ and $(2,2)$ by c. Thus the result of $\mathcal{A} +_{cr} \mathcal{B}$ is model $\mathcal{C}$, which is the expected merge.

Theorem 3.4.6. *[10] If M and N are consistent MTSs then $M +_{cr} N$ is always a common refinement of M and N.*

We now show an example of when the above operator does not produce an LCR. Clearly the merge of a model with itself should result in the same model (i.e.

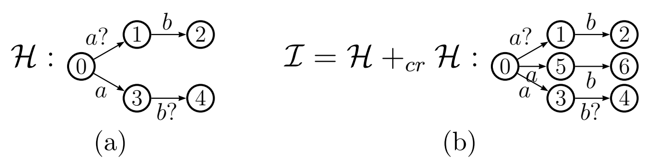


Figure 3.7: Example showing that $+_{cr}$ is not idempotent.

merge should be idempotent) as every model is a refinement of itself and is the least refined one. Consider the following example depicted in Figure 3.7: The result of $\mathcal{H} +_{cr} \mathcal{H}$ is $\mathcal{I}$, which is strictly more refined than $\mathcal{H}$. Note that the conjunction operator produces the same result, i.e. $\mathcal{H} \wedge \mathcal{H} = \mathcal{I}$.

The problem exemplified above arises because $+_{cr}$ does not deal correctly with nondeterminism when there is a mix of required and maybe transitions. Under these circumstances the $+_{cr}$ will apply rules RP and PR, which guarantee to produce a CR but might fail to produce the LCR.

The point is that rules PR and RP of [90, 10] take a conservative decision on how to merge a required with a maybe transition, i.e. that in some cases these rules create a required transition when a maybe transition would suffice. The algorithm we present in Section 3.4.3 is based on detecting the required transitions resulting from these conservative rules which can be converted to maybe transitions.

3.4.2 Consistency Does Not Guarantee Existence of Merge

We have defined the merge between MTSs models as the LCR or one of the MCRs in case there no LCR i.e. there are a non-unique MCRs. Since the merge is defined in terms of minimal common refinements, it is evident that the consistency between models is a necessary condition for the merge to exist.

Otherwise, if the models are not consistent then they do not have any common refinement and therefore the models do not have any MCR. For this reason we have studied and presented a characterisation of consistency in Section 3.3.

An interesting question is whether consistency between models is just a necessary or a sufficient condition for the merge to exist. On different works there is an explicit [90, 10] or implicit [33] assumption that if M and N are consistent MTSs then they have at least one minimal common refinement, i.e. consistency is a sufficient condition for merge. However, in this section we present a counter example where two consistent MTSs do not have any MCRs and therefore the merge is not defined for them.

Proposition 3.4.7 (Consistency does not imply existence of an MCR)**.** *There exist S and T MTSs such that they are consistent but there does not exist any MCR between them.*

Proof. Consider models S and T depicted in Figure 3.4.7. We will first show that the amount of elements of $\mathcal{MCR}(S,T)$ is less or equal to 1, up to equivalence, i.e. $|\mathcal{MCR}(S,T)| \leq 1$. Assume that $\mathcal{MCR}(S,T)$ is not empty, and consider

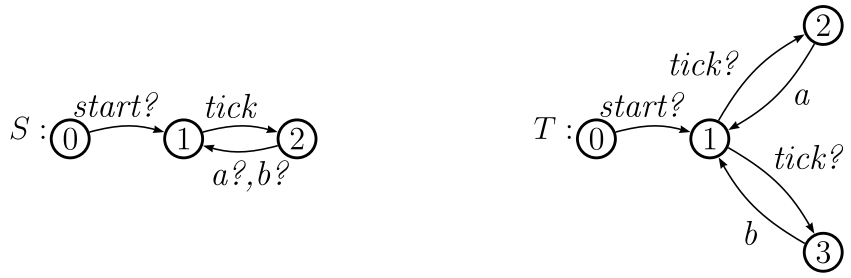


Figure 3.8: Example of two consistent MTSs that do not have any $\mathcal{MCR}$.

$M, N \in \mathcal{MCR}(S,T)$. Since the only transition from the initial state of S and T is a *start*?, it is easy to see that any transitions from the initial states of M or N

must be a *start*?. Then we can define a model W by putting together model M and N and combining their initial state into one. By construction $W \preceq M$ and $W \preceq N$. Considering the way W was constructed, it is trivial to see that $S \preceq W$ and $T \preceq W$, therefore $W \in \mathcal{MCR}(S,T)$. Then by definition of MCR and the fact that $W \preceq M$ and $W \preceq N$, we got that $W \equiv M$ and $W \equiv N$. Therefore $M \equiv N$ which implies that if we assume that $MCR(S,T)$ is not empty then it has only one element up to equivalence, i.e. $|\mathcal{MCR}(S,R)| \leq 1$

If we assume that $\mathcal{MCR}(S,T)$ is not empty, then we know that it can only have one element, so consider that $\mathcal{MCR}(S,T) = M$, with $M = (S, A, \Delta^r, \Delta^p, s_0)$.

- *Observation 1* For each state M_k of M all transitions of M_k have the same label.
- *Observation 2* Each state $M_k \neq M_0$ has at least one required transition.

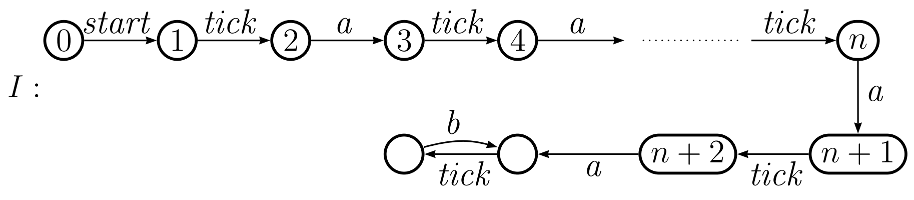


Figure 3.9: Model I is a common implementation of S and T, but it is not an implementation of M.

Let n be the smallest even number greater or equal to the amount of states of M ($n = |S| + |S| \,\%\, 2$). Consider I the MTS depicted in Figure 3.9. Clearly $S \preceq I$ and $T \preceq I$. Since there is a unique MCR then $M \preceq I$. Therefore, there exists an implementation relation R between M and I. I has a sequence of *tick* and a transitions between I_1 and I_{n+2} with $n + 2$ states which is strictly greater than the amount of states of M, thus there exist at least two states on that sequence that are related to the same state of M in R. Let i and j be the largest indexes such that $(I_i, M_x) \in R$ and $(I_j, M_x) \in R$ with $1 \leq i < j \leq n+2$. This implies that there is a loop of length $j - i$ in M of alternating *tick* and a transitions, which is related by R to the states $I_i, I_{i+1}, \ldots, I_j$. Using *observation 1* and *2* it follows that at least all transitions in that loop must be required. But this implies that I should have a loop of alternating *tick* and a transitions which is an absurd that comes from the assumption that there exists an MCR between S and T.

□

On this counter example we presented two consistent models that have an infinite set of non-equivalent common refinements, and that set of common refinements cannot be captured by an MTS. Intuitively the merge of models S and T describes a system where always after a *tick* the system must be able to perform a or b, but not necessarily either of these two actions is required. Informally, the notion that across multiple maybe transitions at least one must be required is something that cannot be captured by an MTS. For example, if we choose to use maybe transitions for a and b then it is impossible to enforce that the system provides at least one of these two actions, leading to undesired implementations. On the other hand, if we use required transitions for a and b then the system is much more restricted than the desired one. There are different extensions to MTS that allow the modeller to express these relations of may and must across multiple transitions like the Disjunctive Modal Transition Systems (DMTS) [63]. In Section 9.1 we provide a survey of related work where we analyse this and other related formalisms.

From an algebraic point of view Proposition 3.4.7 shows that MTSs are not closed with respect to the merge operation. From an engineering perspective this implies that during the elaboration process it might not be possible to construct the merge of two models even though those models are consistent. In this situation we want to provide the modeller with algorithms/tools where he can build a common refinement controlling the complexity of the returned model (e.g. specifying the maximum amount of total states; or giving the number of times a loop, that leads to the inexistence of an MCR, is unrolled). Also, the modeller might decide to elaborate each, or one, of these models further until the model can be merged or the common refinement that can be generated is satisfactory.

3.4.3 A New Merge Algorithm

The algorithm we propose iteratively abstracts the result of $M +_{cr} N$ by replacing required transitions with maybe transitions. It does so while guaranteeing that the resulting MTS after each iteration continues to be a refinement of M and N. The decision of which transitions can be replaced is done by analysing all outgoing required transitions from a given state on a given label. The key notion here is that of *Cover Set*. Intuitively, a cover set describes a set of outgoing required transitions from a given state and on a given label such that if we only keep these as required the model continues to be a common refinement of M and N. Technically, the definition of *cover set* on state c and label ℓ defines a set of states reachable from c on required ℓ transitions such that the required transitions can simulate *all* behaviour starting with ℓ from c.

Definition 3.4.8. (Cover Set) *Let* $A = (S_A, L, \Delta^r_A, \Delta^p_A, a_0)$*,* $B = (S_B, L, \Delta^r_B, \Delta^p_B, b_0)$*,* $C = (S_C, L, \Delta^r_C, \Delta^p_C, c_0)$ *be MTSs such that there exist* R_{AC}*,* R_{BC} *refinement relations between* A *and* C*, and* B *and* C *respectively. Given* $C_i \in S_C$ *and* $\ell \in L$

we define a cover set over C_i on ℓ as a set $\zeta_{C_i,\ell}$ of states of C for which the following holds:

1. $\zeta_{C_i,\ell} \subseteq \Delta^r_C(C_i,\ell)$
2. $\Delta^r_A(R^{-1}_{AC}(C_i),\ell) \subseteq R^{-1}_{AC}(\zeta_{C_i,\ell})$
3. $\Delta^r_B(R^{-1}_{BC}(C_i),\ell) \subseteq R^{-1}_{BC}(\zeta_{C_i,\ell})$

Notation: $\Delta_r(S,\ell) = \{\ t\ |s \xrightarrow{\ell}_{\mathrm{r}} t\ \wedge\ s \in S\}$

Rule 1 states that the cover set $\zeta_{C_i,\ell}$ must be a subset of the states reachable from C_i by single required transitions labelled ℓ. Rule 2 states that the cover set $\zeta_{C_i,\ell}$ preserves in C the required behaviour of A with respect to label ℓ. More precisely, that the set of states in A reachable through required ℓ transitions starting from the states refined by C_i must be a subset of the states in A that are refined by the cover set. In other words, that in C the transitions on ℓ from C_i to the cover set refine all required behaviour on ℓ starting on states refined by C_i. Rule 3 is similar to rule 2, except that it guarantees that the cover set $\zeta_{C_i,\ell}$ preserves in C the required behaviour of B with respect to label ℓ.

As an example of cover set, consider again model $\mathcal{I}$ in Figure 3.7. The cover sets for state 0 and label a are the following: $\{5\}$, $\{3\}$ and $\{3,5\}$ (these sets come from considering the following set of transitions $\{0 \xrightarrow{a} 5\}$, $\{0 \xrightarrow{a} 3\}$, and $\{0 \xrightarrow{a} 3, 0 \xrightarrow{a} 5\}$).

The merge algorithm we propose identifies a cover set for each state s and label ℓ of $A +_{cr} B$ and replaces any required transitions from s on ℓ that is not in the cover set with a maybe transition. We call this MTS transformation an abstraction operation and formalise it as follows:

Definition 3.4.9. (Abstraction operation) *Let A, B, C be MTSs as in Def. 3.4.8. Given a cover set, $\zeta_{C_i,\ell}$, over C_i on ℓ we define the following operation:*

$\mathcal{A}(C, \zeta_{C_i,\ell}) = (S_C, L, \Delta^p_C, \Delta'^r_C, c_0)$ where Δ'^r_C is defined by $\Delta'^r_C = \Delta^r_C - \{(c_i, \ell, c') | c' \notin \zeta_{C_i,\ell}\}$

It is straightforward to show that the abstraction operation on model C effectively produces an MTS that is refined by C. However, it is also the case that it produces a common refinement of A and B.

Theorem 3.4.10. (Abstraction operation produces a CR)
Let A, B, C be MTSs as in Def. 3.4.8 then $A \preceq \mathcal{A}(C, \zeta_{C_i,\ell})$ and $B \preceq \mathcal{A}(C, \zeta_{C_i,\ell})$ for any cover set $\zeta_{C_i,\ell}$

Proof. We want to prove that R_{AC} is a refinement relation between A and $\mathcal{A}(C, \zeta_{C_i,\ell})$. Suppose that this is false, therefore there exists a pair (A', C') in R_{AC} that does not fulfil the refinement relation conditions. If so, we have one possible transition of C' in $\mathcal{A}(C, \zeta_{C_i,\ell})$ that A' cannot simulate; or A' has one required transition, which C' in $\mathcal{A}(C, \zeta_{C_i,\ell})$ cannot simulate with a required transition. The first of these two cases is impossible because $\mathcal{A}(C, \zeta_{C_i,\ell})$ has the same possible transitions as C, and A simulates all possible transitions of C. Therefore the latter must be the case.

Consequently there exists a required transition, $A' \xrightarrow{\ell'}_{\mathrm{r}} A''$, that cannot be simulated by C' in $\mathcal{A}(C, \zeta_{C_i,\ell})$, i.e. $\nexists C'' \cdot C' \xrightarrow{\ell'}_{\mathrm{r}} C'' \wedge (A'', C'') \in R_{AC}$. Since R_{AC} is a refinement relation between A and C, and C and $\mathcal{A}(C, \zeta_{C_i,\ell})$ only differ on the required transitions of C_i on ℓ, then C' must be C_i and ℓ' must be ℓ. Then $(A', C_i) \in R_{AC}$ consequently $A' \in R^{-1}_{AC}(C_i)$, and by the fact that $A' \xrightarrow{\ell'}_{\mathrm{r}} A''$, it follows that $A'' \in \xrightarrow{\ell}_{\mathrm{r}} (R_{AC}^{-1}(C_i))$. Then by definition of cover set it follows that $A'' \in R_{AC}^{-1}(\zeta_{C_i,\ell})$. This implies that

$$\exists C'' \cdot (A'', C'') \in R_{AC}C' \wedge C_i \xrightarrow{\ell'}_{\mathrm{r}} C'' \in \mathcal{A}(C, \zeta_{C_i,\ell})$$

is a contradiction, which comes from the first assumption that R_{AC} is not a refinement relation between A and $\mathcal{A}(C, \zeta_{C_i,\ell})$. As a result we have proved that R_{AC} is a refinement relation between A and $\mathcal{A}(C, \zeta_{C_i,\ell})$

Analogously it can be proven that $B \preceq \mathcal{A}(C, \zeta_{C_i,\ell})$. □

Given that many different cover sets may exist for a specific state and label, the algorithm exploits the following results to select which cover set to use as the basis for applying an abstraction operation and consequently producing a more abstract common refinement. We first define a refinement relation between cover sets which describes a preorder of cover sets based on how refined the application of the abstraction operation using each cover set is. In other words, a cover set is more refined than another if the abstraction operation using the former yields a more refined model than applying the abstraction operation on the latter. We then present a property stating that a refinement between cover sets can be established by checking if all states in one of the cover sets is refined by some state in the other cover set. This property provides an effective way of computing which cover set to use when applying the abstraction operation described above.

Definition 3.4.11 (Cover Set Refinement). *Given $\zeta_{C_i,\ell}$ and $\zeta'_{C_i,\ell}$ cover sets over C_i on ℓ we say that $\zeta_{C_i,\ell}$ is refined by $\zeta'_{C_i,\ell}$, written $\zeta_{C_i,\ell} \preceq \zeta'_{C_i,\ell}$, iff $\mathcal{A}(C, \zeta_{C_i,\ell}) \preceq \mathcal{A}(C, \zeta'_{C_i,\ell})$. Also we might say that $\zeta_{C_i,\ell}$ is more abstract than $\zeta'_{C_i,\ell}$.*

Property 3.4.12. *Given $\zeta_{C_i,\ell}$ and $\zeta'_{C_i,\ell}$ cover sets over C_i on ℓ, $\zeta_{C_i,\ell}$ is refined by $\zeta'_{C_i,\ell}$ if the following condition holds:*

$$\forall C_1 \in \zeta_{C_i,\ell} \cdot \exists C_2 \in \zeta'_{C_i,\ell} \cdot C_1 \preceq C_2$$

We now present the merge algorithm, which starting from $A +_{cr} B$ iteratively applies abstraction operations by identifying for each state and label its least refined cover set.

Algorithm 3.4.13 (Merge algorithm - Initial Attempt)**.**

```
1. $M \leftarrow A +_{cr} B$, $isLCR \leftarrow true$
2. For each $(x, y) \in S_M$ and each $\ell \in Act$ do
   2.1 Get most abstract minimal cover set of $(x, y)$ on $\ell$.
   2.2 If minimal not unique, choose any and
       $isLCR \leftarrow false$.
   2.3 $M \leftarrow \mathcal{A}(M, \zeta_{(x,y),\ell})$
3. Return (M,isLCR)
```

From the results described above it follows that the merge algorithm produces a common refinement of A and B.

Theorem 3.4.14. *The Algorithm 3.4.13 produces a common refinement of A and B.*

Proof. Follows from the fact that $A +_{cr} B$ is a common refinement of A and B and Theorem 3.4.10, which guarantees that every time M is abstracted using operator $\mathcal{A}$ the result is still a common refinement of A and B. □

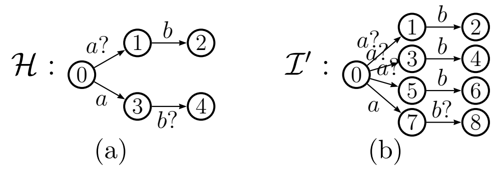


Figure 3.10: Example of a case in which the new merge algorithm outperforms the $+_{cr}$.

We now apply this new algorithm to model $\mathcal{H}$ from our previous example (depicted again in Figure 3.10), where the $+_{cr}$ operator fails to produce the merge. In the same figure we can see model $\mathcal{I}'$, which is the result of applying the merge algorithm to model $\mathcal{H}$ and itself. Model $\mathcal{I}'$ is equivalent to model $\mathcal{H}$. Therefore, on this example the new algorithm produces the optimal solution for the models being merged, improving on the solution given by the $+_{cr}$ and conjunction operators.

The time complexity of the algorithm strongly depends on the amount of non-determinism of the initial model produced by the application of the $+_{cr}$ operation on the input models. If the intermediate model is deterministic the time complexity is polynomial while it grows by an exponential factor based on the degree of non-determinism. We can say that the degree of non-determinism on one state for a particular label is the amount of out transitions with that label minus one. For example, if there are two transitions with the same label going out from one state we say that the degree of non-determinism is one. Furthermore, we say that the degree of non-determinism on a model is the sum of the degree of non-determinism on each possible label on each state.

Although the algorithm presented so far improves the results that can be obtained with the previous algorithms it does not always return the optimal solution, i.e. the merge. This limitation arises when the algorithm finds more than one minimal cover set during the improvement phase. In Figure 3.11 we can see model $\mathcal{G}$, which is the result of the merge algorithm when applied to models $\mathcal{E}$ and $\mathcal{F}$. However, the merge of these two models is $\mathcal{M}$, which is more abstract than $\mathcal{G}$. Intuitively, in this example, when states 1 of both models are merged, model $\mathcal{F}$ imposes that at least one transition by y must be required. But, how does this requirement impact $\mathcal{J}$, which is the $+_{cr}$ of models $\mathcal{E}$ and $\mathcal{F}$? Should the transition that leads to state 2, where maybe a can be taken, be required or the transition to state 4, where maybe b can be taken, be required? Considering the symmetry of the example both options are valid. Requiring both the transition to state 2 and the transition to state 4 to be required is too refined. Requiring any one of them introduces an arbitrary bias towards different sets of implementations. The solution is to clone states in order to allow for both options.

Let us repeat the analysis above using the notion of cover set on model $\mathcal{J}$. The cover sets for state 1 on label y are $\{ 2 \}$, $\{ 4 \}$ and $\{ 2,4 \}$. The first two cover sets are more abstract than the last one, but neither of them refine each other. In this situation, the merge algorithm described above arbitrarily takes one cover set and applies the abstract operator using the chosen cover set. But if we want to build the optimal solution both options must be considered. In other words, the merge should require one of the two y transitions but not necessarily both. This can be achieved by cloning state 1 of model $\mathcal{J}$, having each clone to have one of the cover sets as required transitions, and ensuring that all states that precede state 1 (in this case only state 0) choose non-deterministically over the clones. States 1 and $1'$ in $\mathcal{M}$ are the clones of state 1 in $\mathcal{J}$, where state $1'$ uses cover set $\{ 2 \}$ and state 1 uses cover set $\{ 4 \}$. It is interesting to note that this example also shows that amount of state of the merge can be greater that the amount of states of the $+_{cr}$ between the models.

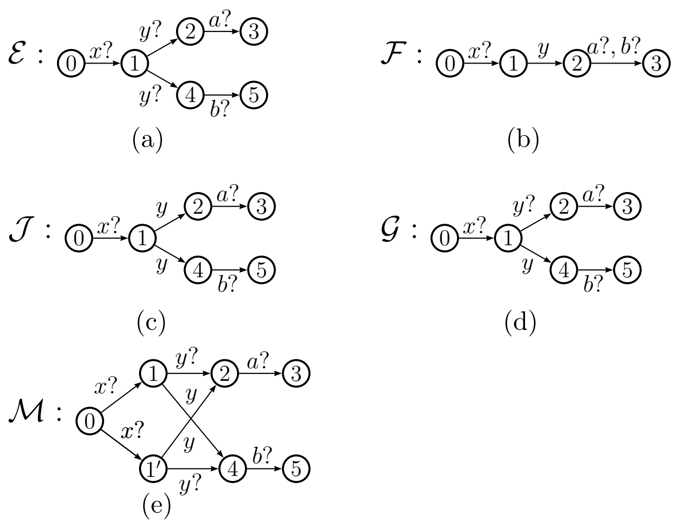


Figure 3.11: Example showing that merge is not bounded by the states of the $+_{cr}$.

We now provide a formal definition of cloning.

Definition 3.4.15. (Clone Operation) *Let* $M = (S, A, \Delta^r, \Delta^p, s_0)$ *be an MTS. For a state* $s \in S$*, let the* clone operation *be defined as* $Clone(M, s) = (S', A, \Delta^{r'}, \Delta^{p'}, s_0)$*, where* $\exists s' \notin S$*, s.t. for* $\ell \in A$*,*

1. $S' = S \cup \{s'\}$
2. $\Delta^{p'} = \Delta^p \cup \{(s', \ell, t) | (s, \ell, t) \in \Delta^p\} \cup \{(t, \ell, s') | (t, \ell, s) \in \Delta^p\}$
3. $\Delta^{r'} = \Delta^r \cup \{(s', \ell, t) | (s, \ell, t) \in \Delta^r\} \cup \{(t, \ell, s') | (t, \ell, s) \in \Delta^r\}$

The following property, which is straightforward to prove, states that the clone operation preserves the semantics of the model.

Property 3.4.16. *The clone operation preserves implementations. In other words,*

$$\mathcal{I}[Clone(M, s)] = \mathcal{I}[M].$$

The final merge algorithm is a variation of the one presented above, in which when incomparable cover sets are identified for a state c, the state is cloned so as to allow applying one cover set on each clone. This algorithm also detects if no MCR exists for the input models by identifying cycles where the degree of non-determinism cannot be reduced after once pass of the abstraction process.

Algorithm 3.4.17. MERGE*(M, N)*

```
Input: consistent MTSs $M = (S_M, Act, \Delta^r_M, \Delta^p_M, s_{0M})$ and
                       $N = (S_N, Act, \Delta^r_N, \Delta^p_N, s_{0N})$
$P \leftarrow M +_{cr} N$
repeat
  $\mathcal{Q} \leftarrow$ emptyQueue
  enqueue(Q, $(s_{0M}, s_{0N})$)
  $\mathcal{V} \leftarrow \emptyset$ //Visited and not Abstracted
  $\mathcal{W} \leftarrow \emptyset$ //Visited and Abstracted
  while $|\mathcal{Q}| > 0$
    $s \leftarrow$ dequeue(Q)
    For each $\ell \in Act$ do
      Let $\mathcal{S}$ be the set of all minimal non-trivial cover sets of $s$ on $\ell$
      if $|S| = 0$
        if $(s, \ell) \in \mathcal{V}$
          continue
        $\mathcal{V} \leftarrow \mathcal{V} \cup \{(s, \ell)\}$
      else
        if $(s, \ell) \in \mathcal{W}$
          abort
        Clone state $s$ in $P$ $|\mathcal{S}| - 1$ times
        For each $i$ do
          take $s_i$ in $S_P$ and $\zeta_{s_i,\ell} \in \mathcal{S}$
          $P \leftarrow Abs(P, \Delta^r_P(s_i, \ell) \setminus \zeta_{s_i,\ell})$
          $\mathcal{W} \leftarrow \mathcal{W} \cup \{(s_i, \ell)\}$
      For each $s'$ such that $(s, \ell, s') \in \Delta^p_P$ do
        enqueue(Q, s')
until no change in $P$
return $P$
```

The correctness of the algorithm given above follows directly from the fact that the clone operation and the abstraction operations are correct.

Theorem 3.4.18. *The Algorithm 3.4.17 produces a common refinement of A and B.*

Proof. Follows from the fact that $A +_{cr} B$ is a common refinement of A and B and Theorem 3.4.10 and Property 3.4.16, which guarantee that after every operation applied to M, the model is still a common refinement of A and B. The algorithm stops since on each iteration the amount of covers sets for each state is reduced. If the algorithm detects that a state that had already been abstracted on the current iteration has increased its number of cover sets, it aborts the process as otherwise it would enter into a loop. This situation arises when model P has a loop than can be unrolled an arbitrary amount of times, always producing a common refinement which is a strict abstraction on the previous model. If model P can be strictly abstracted an arbitrary amount of times then models M and N do not have an MCR since it is always possible to construct a more abstract common refinement. Therefore, the algorithm aborts when detecting this case.

□

An overview of the completeness argument is as follows: When merging consistent models A and B with a unique least common refinement, the only sources of incompleteness of $A +_{cr} B$ (where the model is too refined) are on states with non deterministic choices in which the rules PR or RP have been applied and have generated required transitions when these were unnecessary. Each state c that is a potential source of non-determinism, has the abstraction operation performed on it. The abstraction operation will remove as many required transitions on the non-deterministic choice as possible while preserving refinement of A and B. If there is no unique "best" solution to removing required transitions, the algorithm will perform all incomparable solutions on an equivalent model in which the source of incomparable solutions has been cloned. As this "best" possible abstraction is performed iteratively on all sources of incompleteness, the resulting model is the least common refinement.

As with the previous algorithm, the time complexity is polynomial for deterministic models and grows exponentially with the degree of non-determinism of the models.

From a practical perspective, time complexity of Merge may not be problematic since the algorithm approximates the final result by iterative abstraction operations, and thus the user may decide to cut the process short and he will still obtain a model that is a common refinement of the original ones. As this approximation characterizes implementations that satisfy requirements captured in the original models, it can still be useful for validation and verification of the system behaviour. The only potential issue with cutting the merge of MTSs M and N short is that if the resulting model is then merged with a third model, P, a spurious inconsistency may be obtained: The resulting common refinement of M and N may not be abstract enough to include a valid implementation that is also an implementation of P. This problem can be resolved by computing an n-ary rather than pairwise merge, which we discuss in Section 9.3.

3.4.4 Dealing with Merge When Multiple MCRs Exist

If Algorithm 3.4.17 is applied to a pair of consistent models with multiple MCRs, then the result is one of the MCRs, and the other non-equivalent MCRs are encoded on the returned model as well. The returned model has one initial state denoted by the number 0. If the initial state of the returned model is changed to one of the states denoted by a $0'$, then the other solutions can be produced. The mechanism that deals with the existence of multiple MCRs is that of cloning. If there is a need to clone the initial state of the model produced by $+_{cr}$, and the result of applying on each of the clones the abstraction operation using a different cover set yields non-equivalent states, then the models being merged do not have an LCR.

An example of this can be seen in Figure 3.12. As previously analysed in this section, models $\mathcal{O}$ and $\mathcal{P}$ are consistent and have two minimal common refinements, named $\mathcal{Q}$ and $\mathcal{R}$ (originally depicted in Figure 3.5). Model $\mathcal{S}$ is the result of applying Algorithm 3.4.17 to $\mathcal{O}$ and $\mathcal{P}$. We can see that $\mathcal{S}$ is equivalent to $\mathcal{Q}$, and if we consider $0'$ as the initial state we get a model which is equivalent to model $\mathcal{R}$. Therefore, we have obtained all minimal solutions.

Theorem 3.4.19 (Completeness)**.** *Let A and B be MTSs with the same alphabet and $\mathcal{MCR}(A,B) \neq \emptyset$. Let $M_0 = (S_M, L, \Delta^r_M,\ \Delta^p_M, (a_0,b_0)^0)$ be the result of applying Algorithm 3.4.17 to A and B. Then for any common implementation I of A and B there exists $(a_0,b_0)^j \in S_M$ such that I is an implementation of $M_j = (S_M, L, \Delta^r_M, \Delta^p_M, (a_0,b_0)^j)$.*

This theorem states that for any common implementations I of A and B there is an election of the initial state of M such that I is an implementation of M.

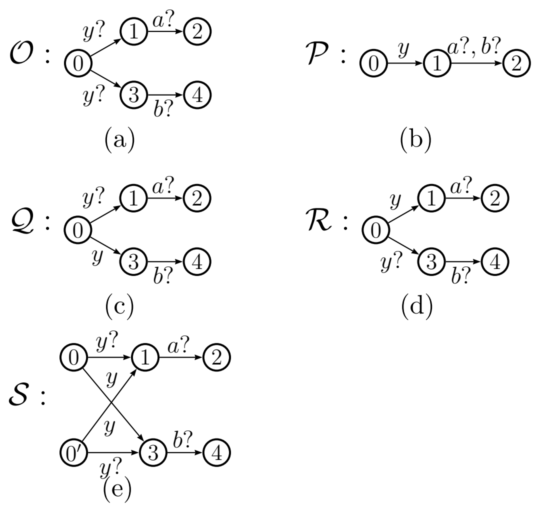


Figure 3.12: Example of a merge where no unique solution exists.

Therefore with Theorem 3.4.18 and 3.4.19 we get that the algorithm returns the *LCR* or a set of *MCRs* that characterise all common implementations if the set of MCRs is not empty.

Corollary 3.4.20. *If Algorithm 3.4.17 does not clone the initial state, i.e.* (a_0, b_0)*, then there exists the LCR of A and B and the algorithm returns it.*

3.4.5 Completeness Proof

We now provide an overview of the proof of the completeness result for Algorithm 3.4.17 as formally stated in Theorem 3.4.19.

We would like to prove that $M_j \preceq I$. In order to do this, we have to show that there is an implementation relation between M_j and I, and that $((a_0, b_0)^j, i_0)$ belongs to that relation. We will do this by introducing auxiliary definitions, presenting Lemmas 3.4.22, 3.4.23 and 3.4.25, and finally proving the theorem using these lemmas.

Since $A \preceq I$, we can take R_{AI} the largest implementation relation between A and I. Analogously, we can take R_{BI}. Let $C_{AB} = \{(a,b) \mid (a,i) \in R_{AI} \wedge (b,i) \in R_{BI}\}$. Let $R_0 = \{((a,b)^c, i) \mid (a,i) \in R_{AI} \ \wedge \ (b,i) \in R_{BI} \ \wedge \ (a,b)^c \in S_M\}$.

Definition 3.4.21. *We say that R* covers *I iff R is a subset of R_0 and has at least one "copy" of each pair (a,b) in C_{AB}, i.e. $R \subseteq R_0$ such that*

$$(\forall a,b,i)\,((a,i) \in R_{AI} \wedge (b,i) \in R_{BI} \implies \exists c\,((a,b)^c, i) \in R)$$

Lemma 3.4.22. *C_{AB} is a consistency relation between A and B.*

Proof. If $a \xrightarrow{\ell}_{\mathrm{r}} a'$ then $i \xrightarrow{\ell} i'$ and $(a',i') \in R_{AI}$. This implies that there exists b' such that $b \xrightarrow{\ell}_{\mathrm{p}} b'$ and $(b',i') \in R_{BI}$. Therefore, b simulates all required behaviour of a with possible transitions and (a',b') belongs to C. Analogously a simulates the required behaviour of b. $\square$

Lemma 3.4.23. *Let R be such that it covers I. Then R is a simulation relation of I by M, i.e. for all $((a,b)^c, i)$ in R the following condition holds:*

$$(\forall \ell, i')(i \xrightarrow{\ell} i' \implies (\exists a', b', c') \cdot ((a,b)^c \xrightarrow{\ell}_{\mathrm{p}} (a',b')^{c'} \ \wedge \ ((a',b')^{c'}, i') \in R))$$

Proof. If $((a,b)^c, i)$ in R then (a,i) belongs to R_{AI}. Therefore, there exists a' such that $a \xrightarrow{\ell}_{\mathrm{p}} a'$ and (a',i') belongs to R_{AI} since I is an implementation of A. Analogously, there exists b' such that $b \xrightarrow{\ell}_{\mathrm{p}} b'$ and (b',i') belongs to R_{BI}. We have $(a',i') \in R_{AI}$, $(b',i') \in R_{BI}$. Using Lemma 3.4.22 we know that (a',b') belongs to $A +_{cr} B$ and $(a,b) \xrightarrow{\ell}_{\mathrm{p}} (a',b')$ in the $+_{cr}$. Then using that R covers I we obtain that there exists c' such that $((a',b')^{c'}, i') \in R)$ and $(a,b)^c \xrightarrow{\ell}_{\mathrm{p}} (a',b')^{c'}$. $\square$

Definition 3.4.24. *Given $R \subseteq R_0$ and $((a,b)^c, i) \in R$ we say that i* simulates *$(a,b)^c$ iff for all a', b', c', ℓ the following condition holds:*

$$(a,b)^c \xrightarrow{\ell}_{\mathrm{r}} (a',b')^{c'} \implies (\exists i')(i \xrightarrow{\ell} i' \wedge ((a',b')^{c'}, i') \in R))$$

Lemma 3.4.25. *For all $(a,i) \in R_{AI}$ and $(b,i) \in R_{BI}$ then there exists c such that i simulates $(a,b)^c$ in R_0.*

Proof. It is enough to show that for $(a,i) \in R_{AI}$ and $(b,i) \in R_{BI}$ and ℓ there exists a minimal cover set $\zeta_{(a,b),\ell}$ of (a,b) by ℓ such that $\zeta_{(a,b),\ell} \subseteq C_{AB}$. Let

$$C = \{(a',b') | i \xrightarrow{\ell} i' \wedge (a',i') \in R_{AI} \wedge (b',i') \in R_{BI}\} \subseteq C_{AB}.$$

It can be easily proved that C is a cover set of (a,b) by ℓ. Now we have to prove that there exists $\zeta_{(a,b),\ell} \subseteq C$, a cover set of (a,b) by ℓ, that is a minimal cover set. We can then choose one of the copies of (a,b) that was generated by the algorithm using $\zeta_{(a,b),\ell}$ as a cover set and that copy is simulated by i since all required transitions from (a,b) go to a state in C. Let $\zeta_m \subseteq C$ be such that for all $\zeta \subseteq C$ cover set, then $\zeta \not\preceq \zeta_m$. Assume that ζ_m is not a minimal cover set. Therefore, there exists ζ a minimal cover set of (a,b) by ℓ such that $\zeta \preceq \zeta_m$ and $\zeta \not\subseteq C$. Thus we can take $(\tilde{a},\tilde{b}) \in \zeta$ such that $(\tilde{a},\tilde{b}) \notin C$, and this also means that $\mathcal{A}(A +_{cr} B, \zeta) \preceq \mathcal{A}(A +_{cr} B, \zeta_m)$. This implies that there exists $(a',b') \in \zeta_m$ with $(a',i') \in R_{AI}$ and $(b',i') \in R_{BI}$ such that (a',b') refines $(\tilde{a},\tilde{b})$ on the models created above. Then we have that i simulates all required behaviour of (a',b'), but since $(\tilde{a},\tilde{b})$ is an abstraction of (a',b'), i must simulate all required behaviour of $(\tilde{a},\tilde{b})$ and therefore $(\tilde{a},\tilde{b})$ belongs to C, which is a contradiction that comes from the assumption that ζ_m is not a minimal cover set. □

(Proof of Theorem 3.4.19). Let

$$R = \{((a,b)^c, i) | (a,i) \in R_{AI} \ \wedge \ (b,i) \in R_{BI} \ \wedge \ i \ simulates \ (a,b)^c\}.$$

We know that $(a_0, i_0) \in R_{AI}$ and $(b_0, i_0) \in R_{BI}$, then by Lemma 3.4.25 we know that there exists j such that i_0 simulates $(a_0, b_0)^j$, which implies that $((a_0, b_0)^j, i_0) \in R$. By Lemma 3.4.25 we can say that R covers I. Then by Lemma 3.4.23 we can say that M_j simulates all behaviour of I, and (by definition of R) I simulates all required behaviour of M_j. Therefore, R is a implementation relation between M_j and I. □

3.4.6 Comparison with Existing Merge Algorithms

It is simple to show that the proposed algorithm produces a more abstract result than [90, 10] as the algorithm starts from the result of $+_{cr}$ and applies zero or more abstraction operations. This entails that for the cases where it is guaranteed that the $+_{cr}$ produces the LCR, the merge algorithm presented herewith will also compute the LCR. For the case of deterministic MTS, our algorithm produces the same result as [90, 10]. In fact, this is also the case for models that satisfy the non-determinacy condition (a slightly weaker condition presented in [90]). This is due to the fact that the non-determinacy condition guarantees that the cover set for every state s and label ℓ contains all required transitions from s on ℓ, and consequently, the abstraction operation becomes idempotent.

The Algorithm 3.4.17 generalizes that of [65] in the sense that it results in the same LCR for any pair of independent MTS, but can also be applied to non-independent yet consistent MTSs producing a common refinement that is the

LCR or encodes all MCRs.

The fact that the merge algorithm does not pick an arbitrary minimal common refinement when merging two models with multiple MCRs is fundamental for supporting elaboration of partial behaviour models. The result of the algorithm can be used to validate each of the MCRs allowing the engineer to make an informed decision on which is the more appropriate merge.

Chapter 4

Weak Alphabet Semantics

Incremental elaboration typically involves gradually extending the scope of a description, i.e. augmenting the alphabet of the models, to describe behaviour aspects that previously had not been taken into account. For example, in our ATM example we may eventually want to describe with a lower level of abstraction the behaviour of the machine during a cash withdrawal operation. This can be seen in Figure 4.1 where the *withdraw* loop-transition on state 2 of model $\mathcal{A}$ in Figure 1.1 has been replaced by a series of transitions in order to describe in more detail the behaviour of the ATM during this operation. In this model, after the user selects *withdraw* she needs to select the amount of money she wants to withdraw. If the user has enough funds available and there is enough cash in the machine, the ATM will dispense the cash. Otherwise, it will display a message indicating to the user the reason for not dispensing the cash.

Once we have extended the alphabet, we will need a way to ensure that the new model does in fact conform to the previous one. We may also need to merge models with different scopes, each one representing a partial description of the complete system.

Existing MTS semantics, strong and weak, require the alphabets of the models being compared to be equal, therefore limiting their applicability in the context of the software development processes we are aiming to support. In order to overcome this limitation we introduce a novel semantics, weak alphabet semantics, which allows the modeller to extend the alphabet of the model as required during the elaboration process to increase the scope of the description. Weak alphabet semantics is based on weak semantics, and so we will first revisit the latter in the following section.

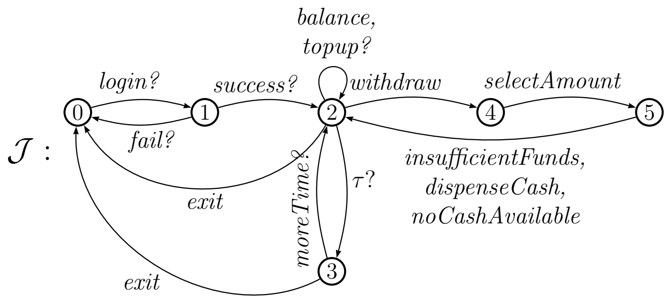


Figure 4.1: A model where the behaviour of functionality associated with a cash withdrawal operation has been detailed.

4.1 Weak Semantics

Weak MTS refinement, defined by Larsen et al. [52], is an observational semantics, i.e. a semantics that treats τ transitions as silent ones. Therefore, it allows for comparing the observable behaviour of models while ignoring the possible differences that they may have in terms of internal computation. Weak MTS refinement is defined based on LTS weak bisimulation, as can be seen in the following definition.

Definition 4.1.1 (Weak Refinement [52]). *A* weak refinement relation *R is a binary relation on δ such that if $(M, N) \in R$ then:*

1. $(\forall \ell, M')(M \xrightarrow{\ell}_{\mathrm{r}} M') \implies (\exists N' \cdot N \overset{\hat{\ell}}{\Longrightarrow}_{\mathrm{r}} N' \ \wedge \ (M', N') \in R)$
2. $(\forall \ell, N')(N \xrightarrow{\ell}_{\mathrm{p}} N') \implies (\exists M' \cdot M \overset{\hat{\ell}}{\Longrightarrow}_{\mathrm{p}} M' \ \wedge \ (M', N') \in R)$

Given MTSs M and N, we say that N is a weak refinement *of M, written $M \preceq_{\mathrm{w}} N$, iff $\alpha M = \alpha N$ and there exists a weak refinement relation R such that $(M, N) \in R$.*

It is worth noting that the relation between weak bisimulation and weak refinement follows the same pattern used to extend strong bisimulation into a refinement. Also, if a model N is a strong refinement of model M ($M \preceq N$) then N is also a weak refinement of M ($M \preceq_{\mathrm{w}} N$). Finally, as with strong refinement, a notion of (weak) implementation can be defined between MTSs and LTSs and refinement is sound with respect to implementation.

The previous definition of weak refinement corresponds to weak modal refinement which, similarly to strong modal refinement, is incomplete. The following definition introduces the notion of thorough weak refinement, defined in terms of inclusion of implementations. This refinement notion, although complete, is computationally expensive and therefore not well suited for our purpose. As stated before, in the rest of this thesis we will be referring to modal refinement unless specifically stated otherwise.

As mentioned before, weak semantics requires the models being considered to have the same communicating alphabet and consequently is not fully adequate to support incremental software development practices. However, it serves as the basis for the definition of a novel semantics, weak alphabet semantics, which allows for extending the alphabet as the modeller needs to increase the scope of the description during model elaboration.

4.2 Weak Alphabet Semantics

4.2.1 Definition

Weak alphabet refinement allows comparing two models in which one has an alphabet that is a superset of the other. This refinement notion aims to capture the intuition of having more information with respect to the common alphabet. It considers all other actions as out of scope for the comparison. To capture this aspect of model elaboration, we first introduce the concept of hiding, an operation that makes a set of actions of a model unobservable to its environment by reducing the alphabet of the model and replacing transitions labelled with an action in the hiding set by τ, as shown below.

Definition 4.2.1 (Hiding)**.** *Let $M = (S, L, \Delta^r, \Delta^p, s_0)$ be an MTS and $X \subseteq Act$. M with actions X hidden, denoted $M \backslash X$, is an MTS $(S, L \backslash X, \Delta^{r'}, \Delta^{p'}, s_0)$, where $\Delta^{r'} = \{(s, \ell, s') \mid \ell \notin X \ \wedge \ (s, \ell, s') \in \Delta^r\} \cup \{(s, \tau, s') \mid \ell \in X \ \wedge \ (s, \ell, s') \in \Delta^r\}$ and analogously for $\Delta^{p'}$. We use $M@X$ to denote $M \backslash (Act \backslash X)$.*

Using the concept of hiding, we now define weak alphabet refinement as an extension of weak refinement in which actions in the extended alphabet are considered unobservable.

Definition 4.2.2. (Weak Alphabet Refinement) *An MTS N is a* weak alphabet refinement *of an MTS M, written $M \preceq_{\mathrm{wa}} N$, iff $\alpha M \subseteq \alpha N$ and $M \preceq_{\mathrm{w}} N@\alpha M$.*

Now consider again model $\mathcal{J}$ depicted in Figure 4.1. If we use weak alphabet refinement to compare it with the initial model $\mathcal{A}$, it can be stated that $\mathcal{J}$ is a refinement of $\mathcal{A}$ being the refinement relation

$$R = \{(0,0), (1,1), (2,2), (2,4), (2,5), (3,3)\}.$$

Thus, as expected, with this semantics the analysed model is an adequate evolution of the initial one.

It is worth noting that weak alphabet refinement is a generalization of weak and strong refinements. In other words, given two models with the same alphabet and no τ-transitions, if one is a strong refinement of the other, then it is also a weak alphabet refinement of the other. Similarly, given two models with the same alphabet but with τ-transitions, if one is a weak refinement of the other, then it is also a weak alphabet refinement of the other.

It is also worth noting that we can have a *thorough* or *modal* notion of weak alphabet refinement depending on whether the underlying notion of weak refinement used in its definition is thorough or modal, respectively. As in the case of strong and weak refinement, thorough weak alphabet refinement is complete albeit computationally expensive while modal weak alphabet refinement is incomplete but can be computed in polynomial time. As mentioned before, in the rest of this thesis if we do not explicitly mention which of these two notions we are referring to, it should be assumed we refer to modal refinement.

If we now go back to the definition of common refinement (Definition 3.4.1), we can see this notion is effectively parameterized by a particular refinement definition, e.g., strong, weak, or weak alphabet. In particular, we can use strong

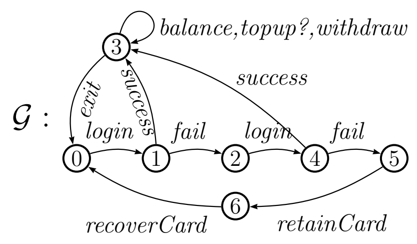


Figure 4.2: An example of weak alphabet common refinement of models $\mathcal{A}$ and $\mathcal{B}$ shown in Figure 1.1.

common refinement when models have the same vocabulary and do not use τ-transitions, weak common refinement when models have the same vocabulary but do use τ-transitions, and weak alphabet common refinement if the alphabets are different. However, for the remainder of this chapter, we assume that common refinement refers to weak alphabet common refinement, unless otherwise specified.

Model $\mathcal{G}$, depicted in Figure 4.2, is an example of a weak alphabet common refinement of models $\mathcal{A}$ and $\mathcal{B}$ shown in Figure 6.1. $\mathcal{G}$ specifies that the ATM must provide two opportunities for logging in, that at the second failed attempt the card is retained, and that once the user is logged in, she can execute several operations. It leaves open whether the ATM should provide a top-up feature. Model $\mathcal{G}$ refines model $\mathcal{A}$ which describes operations to be provided by the ATM to users and also refines model $\mathcal{B}$ which sets the maximum number of failed login attempts to two.

This illustrates how common refinements add required behaviour. Although there is a required transition for withdrawals in model $\mathcal{A}$, this transition is not reachable (through required transitions) from the initial state and thus $\mathcal{A}$ allows implementations in which withdrawals are not possible. However, $\mathcal{B}$ guarantees that implementations will allow succesful logins. Hence, a common refinement of $\mathcal{A}$

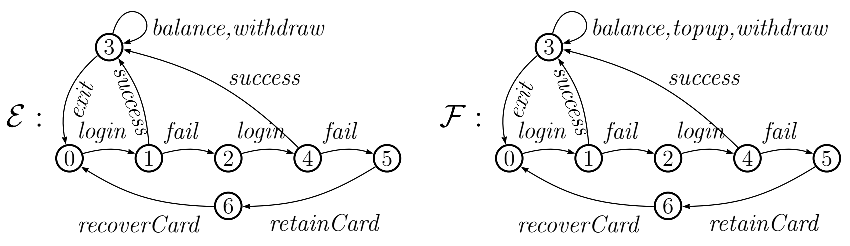


Figure 4.3: Examples of weak alphabet implementations of model $\mathcal{G}$ shown in Figure 4.2.

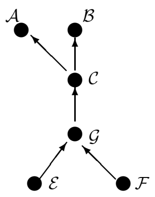


Figure 4.4: Weak alphabet refinement relation between ATM models.

and $\mathcal{B}$, such as $\mathcal{G}$, requires that implementations allow for withdrawals.

Models $\mathcal{E}$ and $\mathcal{F}$ in Figure 4.3 are two examples of implementations of $\mathcal{G}$. We can see that both implementations include the *withdraw* operation.

In Figure 4.4, we depict the alphabet refinement relations that exist between the ATM models previously discussed.

4.2.2 Consistency

In this section, we define consistency relations for weak and weak alphabet semantics and discuss the role of the largest such relation.

In order to merge two consistent models, it is necessary to understand precisely which of their behaviours can be integrated. In particular, a state in any common refinement of two models is intuitively a combination of two consistent states: one from each of the original models. In $M = (S_M, A_M, \Delta^r_M, \Delta^p_M, s_{0M})$ and $N = (S_N,$

A_N, Δ^r_N, Δ^p_N, s_{0N}), states $s \in S_M$ and $t \in S_N$ are consistent if and only if there is a common refinement of M_s and N_t (where M_s indicates changing the initial state of an MTS M to s). Therefore, $N_t@\alpha M$ should be able to simulate required behaviour at M_s with possible behaviour, and vice-versa. A *consistency relation* is used to describe pairs of reachable consistent states.

Definition 4.2.3. (Weak Consistency Relation) *A* weak consistency relation *is a binary relation $C \subseteq \wp \times \wp$, such that the following conditions hold for all $(M, N) \in C$:*

1. $(\forall \ell, M') \cdot (M \xrightarrow{\ell}_{\mathrm{r}} M' \implies \exists N' \cdot (N \overset{\hat{\ell}}{\Longrightarrow}_{\mathrm{p}} N' \wedge (M', N') \in C))$
2. $(\forall \ell, N') \cdot (N \xrightarrow{\ell}_{\mathrm{r}} N' \implies \exists M' \cdot (M \overset{\hat{\ell}}{\Longrightarrow}_{\mathrm{p}} M' \wedge (M', N') \in C))$

The weak consistency relation requires that each model can simulate the required transitions of the other using possible transitions. That is, if M can go to M' on an observable action $\ell \neq \tau$ through a required transition ($M \xrightarrow{\ell}_{\mathrm{r}} M'$), then N can go to N' on a possible transition ($N \overset{\hat{\ell}}{\Longrightarrow}_{\mathrm{p}} N'$) such that M' and N' are consistent. However, N can do so by performing zero or more τ transitions before and after ℓ. On the other hand, if M can move to M' on a τ transition, N can move to N' in zero or more τ moves.

Definition 4.2.4. (Weak Alphabet Consistency Relation) *A* weak alphabet consistency relation *is a binary relation $C \subseteq \wp \times \wp$, such that the following conditions*

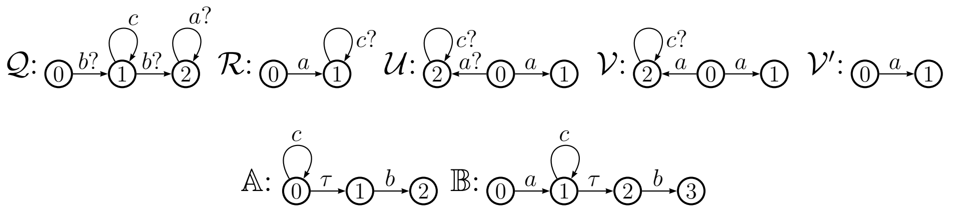


Figure 4.5: Example MTSs for illustrating Weak Alphabet Consistency.

hold for all $(M, N) \in C$*, provided that* $\ell \in Act_\tau$*:*

1. $\forall M' \cdot (M \xrightarrow{\ell}_{\mathrm{r}} M' \;\wedge\; \ell \notin \alpha N \cup \{\tau\}) \implies \exists N' \cdot (N \xRightarrow{\hat{\tau}}_{\mathrm{p}} N' \;\wedge\; (M', N') \in C)$
2. $\forall M' \cdot (M \xrightarrow{\ell}_{\mathrm{r}} M' \;\wedge\; \ell \in \alpha N \cup \{\tau\}) \implies \exists x_1, \ldots, x_n \in (\alpha N \backslash \alpha M) \cdot$
   $\exists N_1, \ldots, N_n, N' \cdot (N \xRightarrow{x_1}_{\mathrm{p}} N_1 \cdots \xRightarrow{x_n}_{\mathrm{p}} N_n \xRightarrow{\hat{\ell}}_{\mathrm{p}} N') \wedge$
   $(\forall i \cdot 1 \leq i \leq n \implies (M, N_i) \in C) \;\wedge\; (M', N') \in C$
3. *Condition 1 defined for* N*.*
4. *Condition 2 defined for* N*.*

The weak alphabet consistency relation is similar in spirit to the weak consistency version (see Definition 4.2.3): a behaviour required in one model must be possible in the other. However, it has two important differences. Firstly, it allows one model to simulate a required ℓ action by performing not only τ's before ℓ, but also any other non-shared action. That is, if M can go to M' through a required transition on a shared action ℓ ($M \xrightarrow{\ell}_{\mathrm{r}} M'$) (antecedent of condition 2 in Definition 4.2.3), then $N@\alpha M$ can simulate ℓ using, if necessary, a succession of possible transitions on actions not in αM. Secondly, it requires that the states traversed by one model to simulate the other preserve the consistency relation. In other words, if $M \xrightarrow{\ell}_{\mathrm{r}} M'$, then all hops $\xRightarrow{x_i}_{\mathrm{p}}$ starting from N before the transition on ℓ (i.e., from N to N_x) must be consistent with M.

For example, consider models $\mathbb{A}$ and $\mathbb{B}$ in Figure 4.5. These models are related by weak alphabet consistency:

$$C_{\mathbb{A}\mathbb{B}} = \{(\mathbb{A}_0, \mathbb{B}_0), (\mathbb{A}_0, \mathbb{B}_1), (\mathbb{A}_1, \mathbb{B}_2), (\mathbb{A}_2, \mathbb{B}_3)\}$$

The transition $\mathbb{A}_0 \xrightarrow{c}_{\mathrm{r}} \mathbb{A}_0$ is simulated by $\mathbb{B}$ with $\mathbb{B}_0 \overset{ac}{\Longrightarrow}_{\mathrm{r}} \mathbb{B}_1$. That is, $\mathbb{B}$ first performs an action that is not observable for $\mathbb{A}$, and then simulates the c action. As we show below (Theorem 4.2.5), the existence of a consistency relation guarantees consistency. Thus, since model $\mathbb{B}$ is a common weak alphabet refinement of itself and model $\mathbb{A}$, these models are consistent.

Consider models $\mathcal{Q}$ and $\mathcal{R}$ in Figure 4.5, where $\alpha\mathcal{R} = \{a, c\}$. There is no weak alphabet consistency relation between them: If there were one, $(\mathcal{Q}_0, \mathcal{R}_0)$ should be in it. As $\mathcal{R}_0 \xrightarrow{a}_{\mathrm{r}} \mathcal{R}_1$, $\mathcal{Q}$ must match this behaviour with $\mathcal{Q}_0 \overset{b}{\Longrightarrow}_{\mathrm{p}} \mathcal{Q}_1 \overset{b}{\Longrightarrow}_{\mathrm{p}} \mathcal{Q}_2 \overset{a}{\Longrightarrow}_{\mathrm{p}} \mathcal{Q}_2$. The definition of weak alphabet consistency requires intermediate state $\mathcal{Q}_1$ to be related to state $\mathcal{R}_0$. But $\mathcal{Q}_1 \overset{c}{\Longrightarrow}_{\mathrm{r}} \mathcal{Q}_1$ and $\mathcal{R}_1$ prohibits c. Hence, assuming a weak alphabet consistency with $(\mathcal{Q}_0, \mathcal{R}_0)$ leads to a contradiction.

The above example illustrates the importance of requiring that intermediate states in the simulation of a by $\mathcal{Q}_0$ be in the consistency relation. If this additional constraint were not included, it would be possible to match $\mathcal{R}_0 \xrightarrow{a}_{\mathrm{r}} \mathcal{R}_1$ with $\mathcal{Q}_0 \overset{a}{\Longrightarrow}_{\mathrm{p}} \mathcal{Q}_2$, constructing the consistency relation between $\mathcal{R}$ and $\mathcal{Q}$. Yet, these models are inconsistent!

The following theorems show the relation between weak and weak alphabet consistency relations and the notion of consistency.

Theorem 4.2.5. (Weak Consistency Relation Characterizes Weak Consistency) *Two MTSs are* weak consistent *iff there is a weak consistency relation between*

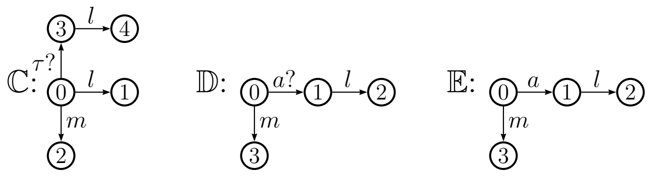


Figure 4.6: MTSs for showing that weak alphabet consistency does not imply existence of a weak alphabet consistency relation between the models.

them.

Theorem 4.2.6. (Weak Alphabet Consistency Relation Entails Weak Alphabet Consistency) *Two MTSs are* weak alphabet consistent *if there is a weak alphabet consistency relation between them.*

Note that the relationship in Theorem 4.2.6 (entailment) is weaker than the one in Theorem 4.2.5 (characterization). The converse of Theorem 4.2.6 does not hold. For example, consider models $\mathbb{C}$ and $\mathbb{D}$ in Figure 4.6. Model $\mathbb{E}$ is their common weak alphabet refinement, so $\mathbb{C}$ and $\mathbb{D}$ are weak alphabet consistent. However, there does not exist a weak alphabet consistency relation between these models: $(\mathbb{C}_0, \mathbb{D}_0)$ must be in the relation and, as $\mathbb{C}_0 \xrightarrow{l}_{\mathrm{r}} \mathbb{C}_1$, so must $(\mathbb{C}_1, \mathbb{D}_2)$ and intermediate state $(\mathbb{C}_0, \mathbb{D}_1)$. However, the latter is clearly inconsistent as $\mathbb{C}_0 \xrightarrow{m}_{\mathrm{r}}$ but this is not the case for $\mathbb{D}_1$ since $\mathbb{D}_1 \not\xrightarrow{m}_{\mathrm{r}}$.

A consistency relation between two models describes consistent behaviours: anything one model does can be simulated by the other. Thus, an interesting and useful consistency relation is the one that captures as much of the consistent behaviour between the models as possible. To describe *all* reachable consistent behaviours between two consistent models, we give the notion of the *largest consistency relation.* It is straightforward to show from Definition 4.2.4 that the union of two consistency relations is also a consistency relation.

Definition 4.2.7. (Largest Consistency Relation) *The* largest consistency relation *between consistent MTSs M and N is*

$$\bigcup\{C_{M,N} \cdot C_{M,N} \textit{ is a consistency relation between } M \textit{ and } N\}.$$

For example, consider models $\mathcal{R}$ and $\mathcal{U}$ in Figure 4.5 defined over the vocabulary $\{\mathsf{a}, \mathsf{c}\}$. While $C_{\mathcal{R}\mathcal{U}} = \{(\mathcal{R}_0,\mathcal{U}_0),\ (\mathcal{R}_1,\mathcal{U}_1),\ (\mathcal{R}_1,\mathcal{U}_2))\}$ is the largest consistency relation between them, $C'_{\mathcal{R}\mathcal{U}} = C_{\mathcal{R}\mathcal{U}} \setminus \{(\mathcal{R}_1,\mathcal{U}_2)\}$ is a consistency relation as well. In particular, these two relations correspond to different common refinements of $\mathcal{R}$ and $\mathcal{U}$, namely, $\mathcal{V}$ and $\mathcal{V}'$ in Figure 4.5. Unlike $\mathcal{V}'$, model $\mathcal{V}$ does not rule out the possibility of an action c occurring after an action a because $C_{\mathcal{R}\mathcal{U}}$ does not exclude the consistent behaviours at $\mathcal{R}_1$ and $\mathcal{U}_2$.

Computing the largest consistency relation between M and N can be done using a fixpoint algorithm, similar to those used for computing bisimulations [34]. Such an algorithm (see Algorithm 4.2.8 below) starts with the Cartesian product of states of MTSs M and N, and then iteratively removes pairs that are not i-step consistent, where i is the number of iterations performed so far.

Algorithm 4.2.8. LargestWeakAlphabetConsistencyRelation*(M, N)*

```
**Input:** MTSs $M = (S_M, A_M, \Delta^r_M, \Delta^p_M, s_{0M})$ and $N = (S_N, A_N, \Delta^r_N, \Delta^p_N, s_{0N})$
$C_0 = \{(M_s, N_t) \cdot s \in S_M \text{ and } t \in S_N\}$
**Repeat**
    $C_{i+1} \leftarrow \{(P, Q) \in C_i \mid (P, Q)$ satisfies conditions 1-4 of Definition 4.2.4$\}$
**Until** $C_{i+1} = C_i$
**Return** $C_i$
```

It is easy to show that the algorithm terminates as it starts from a finite set, C_0, and $C_{i+1} \subseteq C_i$; hence, the greatest fixpoint is reached in at most $\|C_0\|$ steps. Therefore, the time complexity of this algorithm is $O(m \times n^4 \times log(n))$ and the space complexity is $O(n^2)$, where n and m are the maximum number of states and transitions of the input models, respectively. Furthermore, if there exists a consistency relation between M and N then the algorithm returns the largest consistency relation between them.

Theorem 4.2.9. (Soundness of Algorithm 4.2.8) *Let $M = (S_M, A_M, \Delta^r_M, \Delta^p_M, s_{0M})$ and $N = (S_N, A_N, \Delta^r_N, \Delta^p_N, s_{0N})$ be MTSs and C be the relation returned by* CONSISTENCY(M, N). *If $(M, N) \in C$ then C is a weak alphabet consistency relation between M and N.*

The CONSISTENCY algorithm can be used to check whether two models with identical alphabets are consistent (Theorem 4.2.5). However, since the converse of Theorem 4.2.6 does not hold, we cannot rely on this algorithm when it returns false in the case of models with different alphabets. The following result, however, partially resolves this issue by converting the consistency problem between models with different alphabets to a consistency problem between models with identical alphabets.

Theorem 4.2.10. (Consistency Implies Consistency over Common Alphabet) *If M and N are consistent, then $M@(\alpha M \cap \alpha N)$ and $N@(\alpha M \cap \alpha N)$ are consistent as well.*

Hence, if two models are inconsistent w.r.t. their common alphabet, as computed by CONSISTENCY, they are not consistent. Thus, we can determine consistency of models M and N with different alphabets via the following process:

Algorithm 4.2.11. WeakAlphabetConsistent*(M, N)*

```
Input: MTSs $M = (S_M, A_M, \Delta^r_M, \Delta^p_M, s_{0M})$ and $N = (S_N, A_N, \Delta^r_N, \Delta^p_N, s_{0N})$
If $(M, N) \in$ WeakAlphabetConsistencyRelation$(M, N)$
    Return True
$M' \leftarrow M@(\alpha M \cap \alpha N)$
$N' \leftarrow N@(\alpha M \cap \alpha N)$
If $(M', N') \notin$ WeakAlphabetConsistencyRelation$(M', N')$
    Return False
Return Unknown
```

In summary, in this section we have characterized weak (non-alphabet) consistency by means of the existence of a weak consistency relation. In addition, we have shown that the existence of a weak alphabet consistency relation entails the existence of a common weak alphabet refinement. To mitigate the fact that the non-existence of a weak alphabet consistency relation does not entail inconsistency, we have proved a theorem allowing us to relate consistency of models with different alphabets to consistency over their shared alphabet.

4.2.3 Merge

In this section, we describe the algorithm for constructing merge under weak alphabet refinement. We first redefine the $+_{cr}$ operator under weak alphabet semantics and show that if there is a consistency relation between M and N, then $M +_{cr} N$ is a common refinement of M and N. As with strong semantics, the result of $+_{cr}$ may not be an MCR. Hence, we extend the Merge algorithm

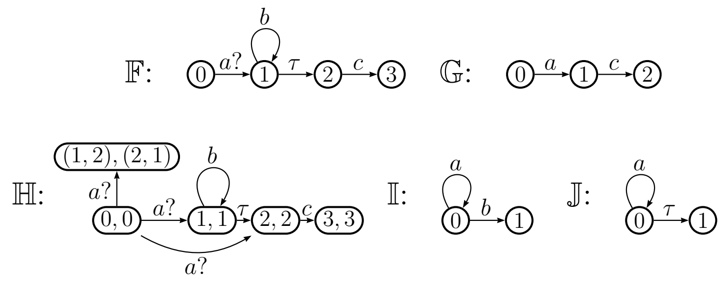


Figure 4.7: Example MTSs for illustrating merge.

(Algorithm 3.4.17) to apply to weak alphabet semantics. This algorithm iteratively abstracts $M +_{cr} N$ while guaranteeing that the result is still a common refinement of M and N, and copes with the case in which two models have more than one MCR.

Building a Common Refinement

In this subsection, we introduce the $+_{cr}$ operator for weak alphabet semantics and show that if there is a consistency relation between M and N, then $M +_{cr} N$ is an element of $\mathcal{CR}(M, N)$, which preserves the properties of the original systems.

Definition 4.2.12. (The $+_{cr}$ operator under weak alphabet semantics) *Let $M = (S_M,\ A_M,\ \Delta^r_M,\ \Delta^p_M,\ s_{0M})$ and $N = (S_N,\ A_N,\ \Delta^r_N,\ \Delta^p_N,\ s_{0N})$ be MTSs and let C_{MN} be the largest weak alphabet consistency relation between them. $M +_{cr} N$ is the MTS $(C_{MN}, A_M \cup A_N, \Delta^r, \Delta^p, (s_{0M}, s_{0N}))$, where Δ^r and Δ^p are the smallest relations that satisfy the rules below, for $\ell \in Act_\tau$:*

$$\textbf{RP}\ \frac{M \overset{\hat{\ell}}{\Longrightarrow}_{\mathrm{r}} M',\ N \overset{\hat{\ell}}{\Longrightarrow}_{\mathrm{p}} N'}{(M,N) \xrightarrow{\ell}_{\mathrm{r}} (M',N')} \qquad \textbf{PR}\ \frac{M \overset{\hat{\ell}}{\Longrightarrow}_{\mathrm{p}} M',\ N \overset{\hat{\ell}}{\Longrightarrow}_{\mathrm{r}} N'}{(M,N) \xrightarrow{\ell}_{\mathrm{r}} (M',N')}$$

$$\textbf{PD}\ \frac{M \overset{\ell}{\Longrightarrow}_{\mathrm{p}} M',\ N \overset{\hat{\tau}}{\Longrightarrow}_{\mathrm{p}} N'}{(M,N) \xrightarrow{\ell}_{\mathrm{r}} (M',N')}\ \ell \notin (\alpha N \cup \{\tau\}) \qquad \textbf{DP}\ \frac{M \overset{\hat{\tau}}{\Longrightarrow}_{\mathrm{p}} M',\ N \overset{\ell}{\Longrightarrow}_{\mathrm{p}} N'}{(M,N) \xrightarrow{\ell}_{\mathrm{r}} (M',N')}\ \ell \notin (\alpha M \cup \{\tau\})$$

$$\textbf{PP}\ \frac{M \overset{\hat{\ell}}{\Longrightarrow}_{\mathrm{p}} M', N \overset{\hat{\ell}}{\Longrightarrow}_{\mathrm{p}} N'}{(M,N) \xrightarrow{\ell}_{\mathrm{p}} (M',N')}$$

Intuitively, the areas of agreement (described by the consistency relation) of the models being combined are traversed simultaneously, synchronizing on shared actions and producing transitions in the resulting model that amount to merging knowledge from both models. Thus, transitions which are possible but not required in one model can be overridden by transitions that are required or prohibited in the other. For example, if M can transit on ℓ through a required transition and N can do so via a possible but not necessarily required transition, then $M +_{cr} N$ can transit on ℓ through a required transition, captured by rules RP and PR in Definition 4.2.12.

The cases in which the models agree on possible transitions are handled by rule PP in Definition 4.2.12. If both M and N can transit on ℓ through possible transitions, then $M +_{cr} N$ can transit on ℓ through a possible transition.

The rules mentioned so far do not apply to non-shared actions. If $\ell \neq \tau$ is not in a model's alphabet, then that model is not concerned with ℓ. Therefore, if the other model can transit on the non-shared action ℓ through a required transition, the merge can do so as well. Rules PD and DP allow the model which does not have ℓ in its alphabet to stay in the same state or to move through τ transitions to another state. The following example motivates this. Consider models $\mathbb{I}$ and $\mathbb{J}$ in Figure 4.7 and assume that $\alpha\mathbb{I} = \{\mathtt{a}, \mathtt{b}\}$ and $\alpha\mathbb{J} = \{\mathtt{a}\}$. The largest consistency relation for $\mathbb{I}$ and $\mathbb{J}$ is $C_{\mathbb{IJ}} = \{(\mathbb{I}_0, \mathbb{J}_0), (\mathbb{I}_1, \mathbb{J}_1)\}$. $\mathbb{I} \xrightarrow{b}_{\mathrm{r}} \mathbb{I}_1$, but $(\mathbb{I}_1, \mathbb{J}_0) \notin C_{\mathbb{IJ}}$ (the above definition requires the resulting model $\mathbb{I} +_{cr} \mathbb{J}$ to stay within consistent states), and therefore, $\mathbb{I}_0 +_{cr} \mathbb{J}_0 \overset{b}{\not\longrightarrow} \mathbb{I}_1 +_{cr} \mathbb{J}_0$. However, $\mathbb{J}_0 \xrightarrow{\tau}_{\mathrm{r}} \mathbb{J}_1$, and $(\mathbb{I}_1, \mathbb{J}_1) \in C_{\mathbb{IJ}}$. Rule PD allows $\mathbb{I} +_{cr} \mathbb{J}$ to have a required transition on $\mathtt{b}$, i.e., $\mathbb{I}_0 +_{cr} \mathbb{J}_0 \xrightarrow{b}_{\mathrm{r}} \mathbb{I}_1 +_{cr} \mathbb{J}_1$. In fact, $\mathbb{I} +_{cr} \mathbb{J}$ is precisely $\mathbb{I}$, which is in $\mathcal{CR}(\mathbb{I},\mathbb{J})$.

Note that rules PD and DP are conservative, i.e., they introduce required rather

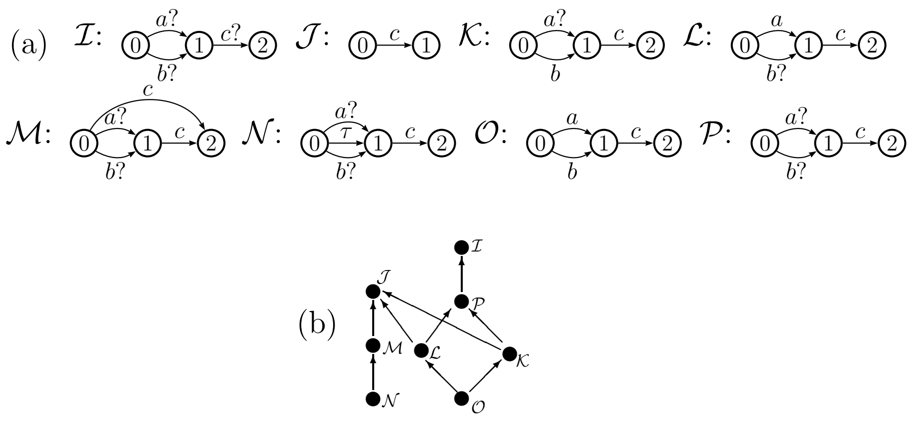


Figure 4.8: (a) Example MTSs. (b) Weak alphabet refinement relation between models of (a).

than possible transitions on ℓ even when neither of the models being composed has a required transition on ℓ. In these rules, if $+_{cr}$ were constructed with possible but not required transitions, then the resulting MTS would not be a common refinement of the models being composed. For instance, considering the models in Figure 4.8(a), $\mathcal{I} +_{cr} \mathcal{J}$ would yield $\mathcal{I}$ (which is not a refinement of $\mathcal{J}$) rather than $\mathcal{O}$.

Special care must be taken in order to combine only consistent behaviours of the two systems (i.e., elements in the consistency relation). For example, suppose that model $\mathbb{F} +_{cr} \mathbb{F}$ (see Figure 4.7) were built without this restriction. There are two transitions on a from the initial state of $\mathbb{F}$, and, therefore, four ways of combining them via the rules in Definition 4.2.12. This composition results in model $\mathbb{H}$, which is *not* a refinement of $\mathbb{F}$. On the other hand, since the pairs $(\mathbb{F}_1, \mathbb{F}_2)$ and $(\mathbb{F}_2, \mathbb{F}_1)$ are not in any consistency relation between $\mathbb{F}$ and itself, constructing $\mathbb{F} +_{cr} \mathbb{F}$ using this restriction yields $\mathbb{F}$, as desired.

When a consistency relation exists, the $+_{cr}$ operator as defined above yields a common refinement of its operands:

Theorem 4.2.13. ($+_{cr}$ builds CRs) *If there is a weak alphabet consistency relation between M and N, then $M +_{cr} N$ is in $\mathcal{CR}(M, N)$.*

For example, suppose we are interested in computing the merge of models $\mathbb{F}$ and $\mathbb{G}$ shown in Figure 4.7, where $\alpha\mathbb{F} = \alpha\mathbb{G} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. The largest consistency relation is $C_{\mathbb{FG}} = \{(\mathbb{F}_0, \mathbb{G}_0), (\mathbb{F}_2, \mathbb{G}_1), (\mathbb{F}_3, \mathbb{G}_2)\}$. Since $\mathbb{F}_0 \overset{a}{\Longrightarrow}_{\mathrm{p}} \mathbb{F}_2$, $\mathbb{G}_0 \overset{a}{\Longrightarrow}_{\mathrm{r}} \mathbb{G}_1$, and $(\mathbb{F}_2, \mathbb{G}_1) \in C_{\mathbb{FG}}$, it follows that $(\mathbb{F}_0, \mathbb{G}_0) \xrightarrow{a}_{\mathrm{r}} (\mathbb{F}_2, \mathbb{G}_1)$ is a transition of $\mathbb{F} +_{cr} \mathbb{G}$ by the PR rule. Since $\mathbb{F}_2 \overset{c}{\Longrightarrow}_{\mathrm{r}} \mathbb{F}_3$, $\mathbb{G}_1 \overset{c}{\Longrightarrow}_{\mathrm{r}} \mathbb{G}_2$, and $(\mathbb{F}_3, \mathbb{G}_2) \in C_{\mathbb{FG}}$, it follows that $(\mathbb{F}_2, \mathbb{G}_1) \xrightarrow{c}_{\mathrm{r}} (\mathbb{F}_3, \mathbb{G}_2)$ is a transition in $\mathbb{F} +_{cr} \mathbb{G}$. Hence, $\mathbb{F} +_{cr} \mathbb{G} = \mathbb{G}$, as desired.

Extending the Merge Algorithm for Weak Alphabet Semantics

While the $+_{cr}$ operator can sometimes produce the LCR, as in the above example, it is generally imprecise. For example, for models $\mathcal{I}$ and $\mathcal{J}$ in Figure 4.8, $\mathcal{I} +_{cr} \mathcal{J} = \mathcal{O}$, but MCRs of $\mathcal{I}$ and $\mathcal{J}$ are $\mathcal{K}$ and $\mathcal{L}$. Since rules DP and PD convert all possible but not required transitions on non-shared actions to required in the composition, thus making the conservative choice, the $+_{cr}$ operator computes a CR that is not necessarily minimal.

Below, we present an extension of the Merge algorithm (Algorithm 3.4.17) for weak alphabet semantics. As mentioned in Section 3.4.3, the algorithm works by detecting the required transitions resulting from the conservative rules of the $+_{cr}$ operator and converting them into possible but not required transitions. It does so while guaranteeing that after each iteration, the resulting MTS continues to be a refinement of the models being merged.

We begin by generalising the abstraction operation so that given an MTS and a subset of its required transitions, it returns an MTS in which these transitions are possible but not required:

Definition 4.2.14. (Abstraction Operation) *Let $M = (S, A, \Delta^r, \Delta^p, s_0)$ be an MTS and let $\Lambda \subseteq \Delta^r$ be a subset of required transitions. Then the* abstraction operation *is defined as follows:*

$$Abs(M, \Lambda) \triangleq (S, A, \Delta^r \setminus \Lambda, \Delta^p, s_0)$$

We now use the abstraction operation to define a more generalised notion of *Cover Set* which considers a set of labels rather than a single label, and uses weak alphabet as the underlying notion of refinement. With this definition, a *Cover Set* is a set of outgoing required transitions from a given state on a given set of labels such that if these are the only transitions kept as required, the resulting model continues to be a common refinement of the models being merged.

Definition 4.2.15. (Cover Set) *Let $M = (S_M,\ A_M,\ \Delta^r_M,\ \Delta^p_M,\ s_{0M})$, $N = (S_N,\ A_N,\ \Delta^r_N,\ \Delta^p_N,\ s_{0N})$ and $P = (S_P,\ A_P,\ \Delta^r_P,\ \Delta^p_P,\ s_{0P})$ be MTSs, with $P \in \mathcal{CR}(M, N)$. For $s \in S_P$ and $A \subseteq A_P$, a set $\zeta_{s,A} \subseteq \Delta^r_P$ is a* cover set *of the state s on labels A iff the following conditions hold:*

1. $\zeta_{s,A} \subseteq \Delta^r_P(s, A)$, *where* $\Delta^r_P(s, A) = \{s \xrightarrow{\ell} s' \in \Delta^r_P \mid \ell \in A)\}$
2. $M \preceq_{\text{wa}} Abs(P, \Delta^r_P(s, A) \setminus \zeta_{s,A})$
3. $N \preceq_{\text{wa}} Abs(P, \Delta^r_P(s, A) \setminus \zeta_{s,A})$

The first rule states that a cover set $\zeta_{s,A}$ of P with respect to M and N is a set of required transitions of P originating from state s on labels in A. The second

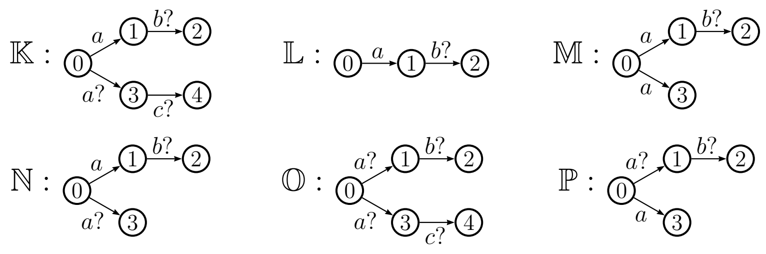


Figure 4.9: Example MTSs for illustrating cover sets.

(third) rule states that if all the required transitions from s on a label in A that do not belong to $\zeta_{s,A}$ are removed, leaving their behaviour as possible but not required, then the resulting MTS is a refinement of M (respectively N).

For example, consider model $\mathbb{M}$ which is a common refinement of models $\mathbb{K}$ and $\mathbb{L}$ (see Figure 4.9). $\zeta_{0,a} = \{0 \xrightarrow{a} 3\}$ is the only non-trivial cover set for $\mathbb{M}$. The result of executing $Abs(\mathbb{M}, \Delta^r_{\mathbb{M}}(0,a) \setminus \zeta_{0,a})$ is model $\mathbb{N}$, which is an abstraction of $\mathbb{M}$ while remaining to be a refinement of $\mathbb{K}$ and $\mathbb{L}$.

Thus, given two models M and N, in order to compute their merge the algorithm should continuously abstract $M +_{cr} N$ while ensuring that the result remains a refinement of M and N, and it seems that the approach to do this is to apply the abstraction operation on cover sets of the common refinement of M and N. However, more than one cover set can exist in this case. For example, consider again models in Figure 4.9. Model $\mathbb{M}$ is a common refinement of models $\mathbb{O}$ and $\mathbb{L}$ and has exactly two non-empty cover sets: $\zeta_{0,a} = \{0 \xrightarrow{a} 1\}$ and $\zeta'_{0,a} = \{0 \xrightarrow{a} 3\}$. The result of $Abs(\mathbb{M}, \Delta^r_{\mathbb{M}}(0,a) \setminus \zeta_{0,a})$ is model $\mathbb{N}$, and the result of $Abs(\mathbb{M}, \Delta^r_{\mathbb{M}}(0,a) \setminus \zeta'_{0,a})$ is model $\mathbb{P}$. While $\mathbb{P}$ is a refinement of $\mathbb{N}$, $\mathbb{N}$ is *not* a refinement of $\mathbb{P}$. Hence, out of the two choices of cover sets, the better one is $\zeta_{0,a}$ since it yields the less refined model. We say that the cover set $\zeta'_{0,a}$ *refines* $\zeta_{0,a}$ (and thus $\zeta_{0,a}$ is the *minimal* cover set). We formalise this intuition below,

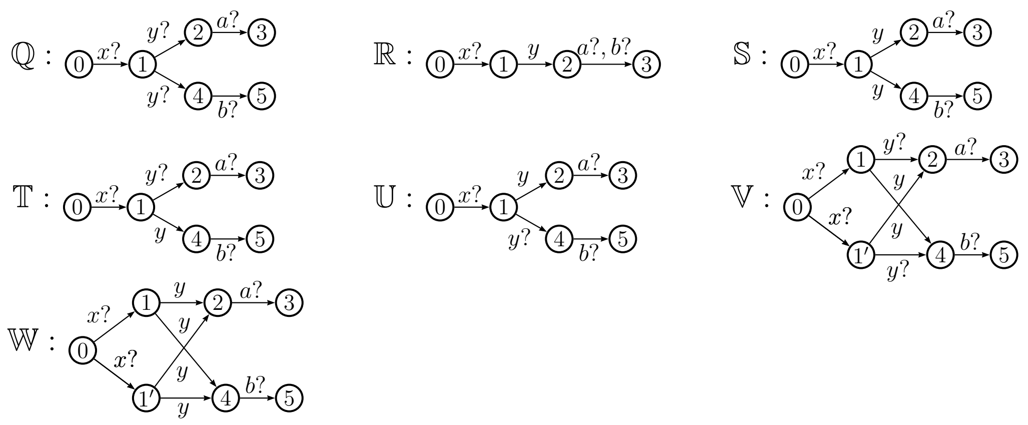


Figure 4.10: Example MTSs for illustrating the need for cloning states.

generalising the cover set refinement definition (Definition 3.4.11) to consider a set of labels rather than a single one.

Definition 4.2.16. (Cover Set Refinement) *Let an MTS $P = (S_P,\ A_P,\ \Delta^r_P,\ \Delta^p_P,\ s_{0P})$ be given and let $A \subseteq A_P$. For a pair of cover sets over a state s on A, $\zeta_{s,A}$ and $\zeta'_{s,A}$, we say that $\zeta_{s,A}$* is refined by *$\zeta'_{s,A}$, written $\zeta_{s,A} \preceq \zeta'_{s,A}$, iff $Abs(P, \Delta^r_P(s, A) \setminus \zeta_{s,A}) \preceq Abs(P, \Delta^r_P(s, A) \setminus \zeta'_{s,A})$.*

As expected, refinement of cover sets defines a preorder, i.e. a common refinement may have two cover sets where neither refines the other. Consider the models in Figure 4.10. Model $\mathbb{S}$, a common refinement of $\mathbb{Q}$ and $\mathbb{R}$, has exactly two non-empty cover sets: $\zeta_{1,y} = \{1 \xrightarrow{y} 2\}$ and $\zeta'_{1,y} = \{1 \xrightarrow{y} 4\}$. Neither of these cover sets refine each other as $Abs(\mathbb{S}, \Delta^r_{\mathbb{S}}(1, y) \setminus \zeta_{1,y})$ (model $\mathbb{U}$) is not a refinement of $Abs(\mathbb{S}, \Delta^r_{\mathbb{S}}(1, y) \setminus \zeta'_{1,y})$ (model $\mathbb{T}$), nor is the latter a refinement of the former. As analysed in Chapter 3, an algorithm that picks only one of these cover sets to abstract $\mathbb{S}$ is not able to compute the LCR of $\mathbb{Q}$ and $\mathbb{R}$: model $\mathbb{V}$. To compute $\mathbb{V}$ from $\mathbb{S}$, we need to clone state 1 in model $\mathbb{S}$, obtaining an equivalent model, model $\mathbb{W}$, which allows an application of a different cover set for each copy of

Algorithm 4.2.17. Merge*(M, N)*

```
Input: consistent MTSs $M = (S_M, A_M, \Delta^r_M, \Delta^p_M, s_{0M})$ and $N = (S_N, A_N, \Delta^r_N, \Delta^p_N, s_{0N})$
$P \leftarrow M +_{cr} N$
$\mathcal{A} \leftarrow \{\{\ell\} \mid \ell \in (\alpha M \cap \alpha N)\} \cup \{(\alpha M \setminus \alpha N) \cup (\alpha N \setminus \alpha M) \cup \{\tau\}\}$
repeat
    $\mathcal{Q} \leftarrow$ emptyQueue
    enqueue(Q, $(s_{0M}, s_{0N})$)
    $\mathcal{V} \leftarrow \emptyset$ //Visited and not Abstracted
    $\mathcal{W} \leftarrow \emptyset$ //Visited and Abstracted
    while $|\mathcal{Q}| > 0$
        $s \leftarrow$ dequeue(Q)
        For each $A \in \mathcal{A}$ do
            Let $\mathcal{S}$ be the set of all minimal non-trivial cover sets of $s$ on $A$
            if $|S| = 0$
                if $(s, A) \in \mathcal{V}$
                    continue
                $\mathcal{V} \leftarrow \mathcal{V} \cup \{(s, A)\}$
            else
                if $(s, A) \in \mathcal{W}$
                    abort
                Clone state $s$ in $P$ $|\mathcal{S}| - 1$ times
                For each $i$ do
                    take $s_i$ in $S_P$ and $\zeta_{s_i,A} \in \mathcal{S}$
                    $P \leftarrow Abs(P, \Delta^r_P(s_i, A) \setminus \zeta_{s_i,A})$
                    $\mathcal{W} \leftarrow \mathcal{W} \cup \{(s_i, A)\}$
            For each $s'$ such that exists $\ell \in A \cdot (s, \ell, s') \in \Delta^p_N$ do
                enqueue(Q, s')
until no change in $P$
return $P$
```

Figure 4.11: The Merge algorithm.

state 1 in order to abstract the model.

We are now ready to present the Merge algorithm for weak alphabet semantics (see Algorithm 4.2.17 in Figure 4.11). As illustrated earlier in this section, the Merge algorithm computes a common refinement of two consistent models and then iteratively abstracts it by abstracting required transitions based on least refined cover sets of the common refinement. Should there be more than one, the algorithm clones the appropriate states and applies abstraction with respect to each cover set to each clone.

When applied to models $\mathbb{Q}$ and $\mathbb{R}$ in Figure 4.10, this algorithm yields model $\mathbb{V}$, as desired.

The algorithm includes one small optimization: rather than looking for cover sets for all possible subsets of $\alpha M \cup \alpha N$, it only builds cover sets for singleton sets over the common alphabet of M and N (i.e., $\{\ell\} \mid \ell \in (\alpha M \cap \alpha N)$) and the set of actions that are not observable to either M or N (i.e., $(\alpha M \setminus \alpha N) \cup (\alpha N \setminus \alpha M) \cup \{\tau\}$). This is because any other subset of $\alpha M \cup \alpha N$ will, by definition of cover set and refinement, never yield a cover set.

Termination of the algorithm is guaranteed as each iteration considers a fewer number of cover sets for the current state and its clones. Otherwise, an abort statement is invoked. The correctness of this algorithm is straightforward to prove using properties of cloning and cover sets. The latter are by definition guaranteed to result in a common refinement when used in the context of an abstraction operation.

As we discussed in Section 3.4.2, two consistent models may have a unique least common refinement, a set of non-equivalent minimal common refinements, or there may not exist any minimal common refinement at all. When there exists a consistency relation between the models being merged the algorithm deals with all these three cases correctly and effectively, returning the LCR if the LCR exists, a model that encodes all MCRs if there are multiple MCRs, or aborting if the merge is not defined for the given models.

However, unlike merge under strong semantics, the MERGE algorithm for weak alphabet semantics is not complete. The reason for incompleteness is that in the case of weak alphabet semantics, the non-existence of a consistency relation does not imply inconsistency. Hence, given two consistent models, the MERGE algorithm may not execute because a consistency relation does not exist. This is not a limitation if the two models being merged have the same alphabet, possibly with τ transitions, as the notion of consistency relation is complete with respect to weak semantics. In addition, the algorithm presented in this section is an extension of Algorithm 3.4.17 and hence is complete with respect to strong semantics.

While incomplete, the MERGE algorithm improves on the one presented in [90]. For the cases handled by the algorithm in [90], MERGE can compute more abstract common refinements. In addition, MERGE can compute minimal common refinements for a broader range of consistent MTSs.

The MERGE algorithm for weak alphabet semantics has the same complexity as the merge algorithm for strong semantics. However, while both algorithms are exponential in time and polynomial in space, the degree of non-determinacy tends to be higher for the cases of weak alphabet refinement, rendering the corresponding algorithm more expensive in practice.

Chapter 5

Branching Alphabet Semantics

5.1 Motivation

One of the problems of weak MTS semantics is that it allows implementations that can be considered unintuitive. Consider the MTS $\mathcal{K}$ in Figure 5.1, which is a valid implementation of model $\mathcal{A}$ of the ATM (Figure 1.1) based on the weak implementation relation

$$R = \{(0,0),(1,1),(2,2)\}.$$

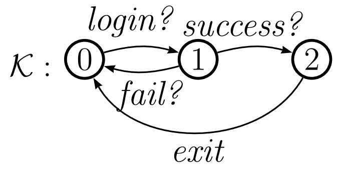


Figure 5.1: A valid implementation of model $\mathcal{A}$ according to weak refinement.

Note that in $\mathcal{A}$ the exact details of the login process, such as the maximum number of failed login attempts, are yet to be defined, but if the user succeeds on the login we would expect the *balance*, *withdraw* and optionally the *topup* functions to be reachable for the user. However, in the implementation proposed above the user never has the possibility of selecting any of these functionalities after successfully logging in. This implementation does not reflect the expected behaviour for the system, thus breaking the intuition behind the notion of implementation. This example shows that weak semantics does not seem to be adequate to support evolving software modelling since it accepts as valid refinements counter intuitive implementations. In subsequent sections we shall also show that weak semantics lacks some properties that relate refinement with action hiding. These properties are linked to some degree to the existence of such unintuitive implementations that weak semantics allows.

To gain some insight as to why weak refinement leads to such unintuitive implementations an alternative (yet equivalent) standard definition of weak refinement can be used. Weak refinement can be thought of as simply applying strong refinement to the models obtained from performing the transitive closure of τ transitions (Property 5.1.2).

Definition 5.1.1 (Observational Graph). *Given an MTS $M = (S, L, \Delta^r, \Delta^p, s_0)$, the* observational graph *of M is the derived MTS $Obs(M) = (S, L, \Delta^r_o, \Delta^p_o, s_0)$ where Δ^r_o, Δ^p_o are given by:*

$$\Delta^r_o = \{M \xrightarrow{\ell} M' \mid M \overset{\hat{\ell}}{\Longrightarrow}_{\mathrm{r}} M'\}$$
$$\Delta^p_o = \{M \xrightarrow{\ell} M' \mid M \overset{\hat{\ell}}{\Longrightarrow}_{\mathrm{p}} M'\}$$

Definition 5.1.2 (Weak Refinement). *N is a weak refinement of M, written $M \preceq_{\mathrm{w}} N$, if and only if $Obs(M) \preceq Obs(N)$.*

We now revisit the example with this alternative definition of weak semantics. The transitive closure of τ performed on model $\mathcal{A}$ yields the model in Figure 5.2. The transition labelled *success* in model $\mathcal{A}$ gives rise to two different maybe transitions in Figure 5.2, i.e. $1 \xrightarrow{success}_{\mathrm{m}} 2$ and $1 \xrightarrow{success}_{\mathrm{m}} 3$. To obtain the implementation in Figure 5.1, we can consider that these two transitions were refined differently: $1 \xrightarrow{success}_{\mathrm{m}} 3$ became a required behaviour while $1 \xrightarrow{success}_{\mathrm{m}} 2$ became proscribed. This kind of "inconsistent" decisions that weak refinement allows over different transitions in the closured model, that were originated from the same maybe transition in the original model, are the cause for these unexpected implementations.

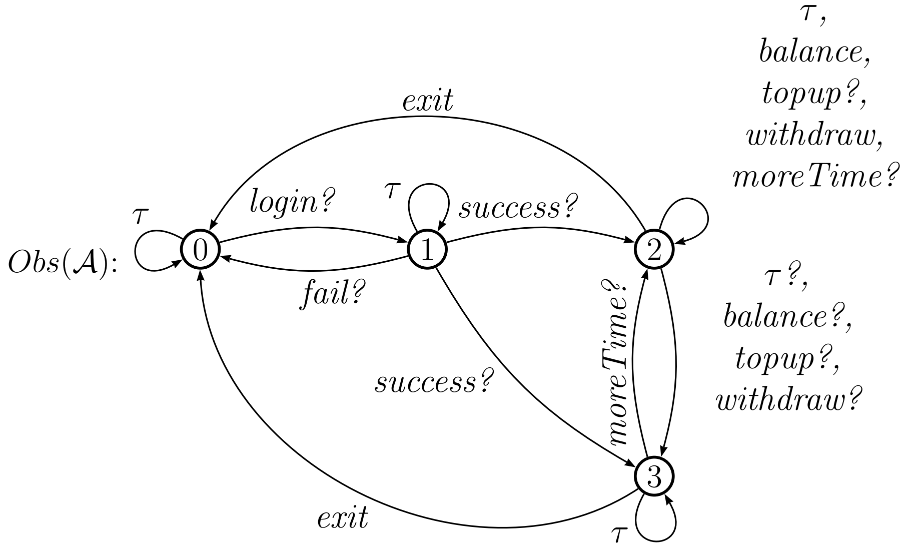


Figure 5.2: Observational graph of model $\mathcal{A}$ in Figure 1.1.

So far we have seen that although an observational semantics is required to support incremental elaboration of partial behaviour models, the observational semantics based on weak refinement might not adequately reflect the intended meaning of an MTS. In the next section we introduce a semantics that not only resolves the case discussed above but also provides a number of theoretical results that support the argument for a novel observational semantics for MTS.

5.2 Branching Semantics

We have analysed the shortcomings of strong and weak semantics for our modelling purposes. The former semantics does not distinguish observable from unobservable actions and hence does not support elaboration when varying the level of abstraction of partial models. The latter does not preserve branching behaviour adequately, therefore allowing implementations that contradict the intuition users may have of partial models. In the rest of this chapter, we will first explore the intuition we have for MTS semantics by analysing specific examples and then formally define a novel semantics that captures this intuition.

5.2.1 Exploration

We have already settled that the new semantics should be an observational one that also preserves the branching structure. In this section we present a series of examples that will allow us to further clarify the behaviour we are looking for in the new semantics. Each of the examples consists of a model M and a model N and all of them fulfil the following condition: according to weak semantics N is considered to be a refinement of M, whereas according to strong semantics no refinement relation can be defined between them. In each case, we are going to state whether or not we expect the new semantics to define a refinement relation between the two models and explain the reasons that support this choice.

$$\begin{tikzcd}
\mathcal{M}_1 : 0 \arrow[r, "\tau?"] \arrow[d, "b"] & 1 \arrow[r, "a"] & 2 & \not\preceq & \mathcal{N}_1 : 0 \arrow[r, "a?"] \arrow[d, "b"] & 1 \\
3 & & & & 2 &
\end{tikzcd} \tag{5.1}$$

Consider example (5.1). In this case it can be seen that a valid implementation of $\mathcal{N}_1$ is *(a+b)*. However, we do not want that implementation to be a valid one for $\mathcal{M}_1$ since intuitively this would not preserve the branching structure. If we look at model $\mathcal{M}_1$ we will realize that in order to take a we should always reach a state from where a can be taken but b cannot. Thus, allowing *(a+b)* as a possible implementation would violate the branching structure and so in the new semantics $\mathcal{N}_1$ should not constitute a refinement of $\mathcal{M}_1$.

$$\begin{tikzcd}
 & & 2 & & & 1 \\
\mathcal{M}_2 : 0 \arrow[r, "\tau?"] & 1 \arrow[ur, "a"] \arrow[dr, "b"'] & & \not\preceq & \mathcal{N}_2 : 0 \arrow[ur, "a?"] \arrow[dr, "b"'] & \\
 & & 3 & & & 2
\end{tikzcd} \tag{5.2}$$

In example (5.2) our intention with model $\mathcal{M}_2$ is to describe that either the system cannot take any observable transition or it can take both a and b *(a+b)*. On the other hand model $\mathcal{N}_2$ allows b as a possible implementation and hence we do not want $\mathcal{N}_2$ to be a refinement of $\mathcal{M}_2$ in the new semantics.

$$\begin{tikzcd}
 & & & 2 & & & \\
\mathcal{M}_3 : & 0 \arrow[r, "\tau?"] \arrow[d, "b?"] & 1 \arrow[ur, "a"] \arrow[dr, "b"] & & \not\preceq \ \mathcal{N}_3 : & 0 \arrow[r, "a?"] \arrow[d, "b?"] & 1 \\
 & 4 & & 3 & & 2 &
\end{tikzcd} \tag{5.3}$$

Model $\mathcal{N}_3$ allows a as a possible implementation. However, model $\mathcal{M}_3$ has a structure such that if transition a can be taken then transition b is also possible. For this reason we do not want the new semantics to consider model $\mathcal{N}_3$ to be a refinement of model $\mathcal{M}_3$.

$$\begin{tikzcd}
 & & & & 3 & & & \\
\mathcal{M}_4 : & 0 \arrow[r, "\tau"] \arrow[d, "b"] & 1 \arrow[r, "\tau?"] & 2 \arrow[ur, "a"] \arrow[dr, "b"] & & \not\preceq \ \mathcal{N}_4 : & 0 \arrow[r, "a?"] \arrow[d, "b"] & 1 \\
 & 5 & & & 4 & & 2 &
\end{tikzcd} \tag{5.4}$$

Model $\mathcal{M}_4$ is different from model $\mathcal{M}_3$ in that it adds a new required τ transition in such a way that any implementation that aims to preserve the branching structure should have two branches. Therefore, b is not a valid implementation of this model. Since $\mathcal{N}_4$ accepts this implementation as a valid one, we expect the new semantics not to consider $\mathcal{N}_4$ as a refinement of $\mathcal{M}_4$.

Let's consider now the following example:

$$\begin{tikzcd}
 & & & 3 & & 1 \\
\mathcal{M}_5 : 0 \arrow[r, "\tau?"] & 1 \arrow[r, "\tau?"] & 2 \arrow[ur, "a"] \arrow[dr, "b"'] & \preceq & \mathcal{N}_5 : 0 \arrow[ur, "a"] \arrow[dr, "b"'] & \\
 & & & 4 & & 2
\end{tikzcd} \tag{5.5}$$

Model $\mathcal{N}_5$ is a weak implementation of model $\mathcal{M}_5$ and, since obviously the latter preserves the structure of the former, we would like the new semantics to also consider model $\mathcal{N}_5$ as an implementation of $\mathcal{M}_5$.

$$\begin{tikzcd}
 & & & 3 & & 1 \\
\mathcal{M}_6 : 0 \arrow[r, "\tau?"] \arrow[d, "b?"] & 1 \arrow[r, "\tau?"] \arrow[d, "c"] & 2 \arrow[ur, "a"] \arrow[dr, "b"'] & \npreceq & \mathcal{N}_6 : 0 \arrow[ur, "a?"] \arrow[dr, "b"'] & \\
5 & 6 & & 4 & & 2
\end{tikzcd} \tag{5.6}$$

Finally, consider example (5.6) where model $\mathcal{N}_6$ accepts *(a+b)* as a valid implementation. If we analyse model $\mathcal{M}_6$ we will see that intuitively any valid implementation of this model that includes transition a should also include the possibility of taking transition c before. This is another example that depicts how weak semantics allows for refinements that contradict the intuitive idea of elaboration of partial models since it does not preserve branching behaviour adequately.

5.2.2 Definition

In this section we define a new semantics that has the desired properties of both weak and strong semantics, i.e. an observational semantics that preserves branching structure. To do this we consider a third equivalence over LTSs called branching equivalence which, as we have already seen in the background, can be situated between strong and weak equivalences.

As shown previously, the three equivalences on LTSs are given by a symmetric simulation relation, the difference between them is the way a transition on one model is simulated by the other model. Figure 5.3 shows a graphical representation of how a transition is simulated in each of these three equivalences. We have also shown strong and weak MTS refinements to be loosely based on the corresponding LTS equivalences by having a slightly asymmetric bisimulation in that every required transition in the less refined model must be simulated by the refined model using only required transitions, and every possible transition in the refined model must be simulated by possible transitions of the less refined model.

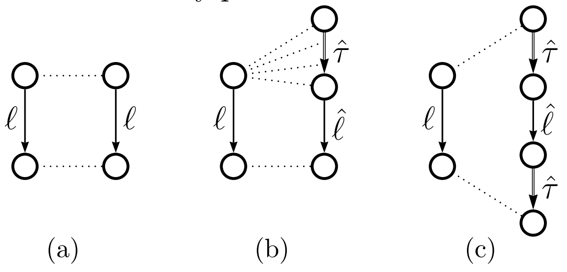


Figure 5.3: Depiction of how a transition $\bigcirc \xrightarrow{\ell} \bigcirc$ is simulated in bisimulation: (a) strong; (b) branching; (c) weak.

LTS branching bisimulation cannot be adapted in a similar way to produce an adequate MTS refinement. Definition 5.2.1 presents the refinement that would result from doing so.

Definition 5.2.1 (Naïve branching refinement relation)**.** *A* naïve branching refinement relation *R is a binary relation on δ such that if $(M, N) \in R$ then:*

1. $(\forall \ell, M')(M \xrightarrow{\ell}_{\mathrm{r}} M') \implies (\exists N', N'' \cdot N \overset{\tau}{\Longrightarrow}_{\mathrm{r}} N' \xrightarrow{\hat{\ell}}_{\mathrm{r}} N'' \ \wedge\ (M, N'), (M', N'') \in R)$
2. $(\forall \ell, N')(N \xrightarrow{\ell}_{\mathrm{p}} N') \implies (\exists M', M'' \cdot M \overset{\tau}{\Longrightarrow}_{\mathrm{p}} M' \xrightarrow{\hat{\ell}}_{\mathrm{p}} M'' \ \wedge\ (M', N), (M'', N') \in R)$

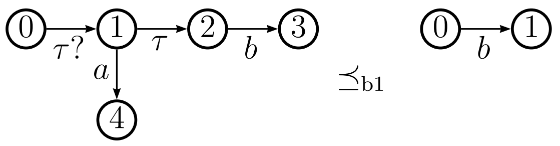


Figure 5.4: Example of a refinement according to Definition 5.2.1, where the branching structure of the less refined process is not preserved.

If we said that N is a naïve branching refinement of M if $\alpha M = \alpha N$ and (M, N) is contained in some branching refinement relation R, the above definition would not lead to an adequate refinement notion since it does not preserve branching structure. Figure 5.4 shows an example of a model refining another model without preserving the branching structure;

$$R = \{(0,0), (2,0), (3,1)\}$$

is the refinement relation between these models according to the previous definition.

The reason why this definition does not preserve the branching structure is that it does not guarantee that all intermediate states of $M \overset{\tau}{\Longrightarrow}_{\mathrm{p}} M'$ are related to N, as the stuttering lemma (2.1.5) states for branching equivalence. To amend this problem the previous definition needs to be reinforced by explicitly requiring that all intermediate states reached by τ transitions have to be in the relation, enforcing the stuttering property, as shown in the following definition:

Definition 5.2.2 (Branching refinement relation)**.** *A branching refinement rela-*

tion R *is a binary relation on* δ*, such that if* $(M, N) \in R$ *then:*

1. $(\forall \ell, M')(M \xrightarrow{\ell}_{\mathrm{r}} M') \implies (\exists\, N_0, \ldots, N_n, N' \cdot$
   $N_i \xrightarrow{\tau}_{\mathrm{r}} N_{i+1}\ \forall\, 0 \leq i < n\ \wedge\ N_n \xrightarrow{\hat{\ell}}_{\mathrm{r}} N' \wedge$
   $N_0 = N\ \wedge\ (M, N_i) \in R\ \forall\, 0 \leq i \leq n \wedge$
   $(M', N') \in R)$
2. $(\forall \ell, N')(N \xrightarrow{\ell}_{\mathrm{p}} N') \implies (\exists\, M_0, \ldots, M_n, M' \cdot$
   $M_i \xrightarrow{\tau}_{\mathrm{p}} M_{i+1}\ \forall\, 0 \leq i < n\ \wedge\ M_n \xrightarrow{\hat{\ell}}_{\mathrm{p}} M' \wedge$
   $M_0 = M\ \wedge\ (M_i, N) \in R\ \forall\, 0 \leq i \leq n \wedge$
   $(M', N') \in R)$

Let us consider that N is a branching refinement of M if $\alpha M = \alpha N$ and (M, N) is contained in some branching refinement relation R. Then this definition will preserve the branching structure, since every intermediate state a model goes through when simulating a transition on the other model is actually related to the initial state of that transition. Intuitively, this means that none of those intermediate states present more or less behaviour than the initial state. In particular, it solves the problem depicted in Figure 5.4.

However, this modal definition would not induce a complete semantics. In other words, if we consider the set of implementations of M and N the fact that one is included in the other one does not imply the existence of a refinement relation between them. Figure 5.5 shows an example of this case. Although the two models shown have the same set of possible implementations according to Definition 5.2.2, there is no appropriate refinement relation for $A1 \preceq_{\mathrm{b}} A2$. This result is expected if we note that strong modal refinement is finer than the modal refinement we have just defined, and as shown in section 3.2 strong modal refinement does not induce a complete semantics.

$$\begin{tikzcd} A1: \textcircled{0} \arrow[r, "\tau?"] & \textcircled{1} \arrow[r, "a"] & \textcircled{2} & \not\preceq_{\mathrm{b}} & A2: \textcircled{0} \arrow[r, "a?"] & \textcircled{1} \end{tikzcd}$$

Figure 5.5: Models $A1$ and $A2$ have the same set of implementations according to Definition 5.2.2 but one is not a branching modal refinement of the other according to the same definition.

To overcome these limitations, and recalling that an MTS semantics is completely defined by stating which are valid implementations for a model, we define branching thorough refinement based on inclusion of implementations. The notion of branching thorough refinement comes naturally as N is a refinement of M if all the branching implementations of N are branching implementations of M. Branching modal refinement can be seen as an operation that approximates thorough refinement. As with strong and weak refinement, the modal refinement can be computed in polynomial time, while thorough refinement is computationally more expensive. Although we have not studied the complexity of computing branching thorough refinement, a lower bound is given by the complexity of computing strong thorough refinement, which is EXPTIME-complete [7].

We now formalise the definition of branching implementation. Recall that $\wp$ is the universe of all LTSs and δ of all MTSs.

Definition 5.2.3 (Branching Implementation)**.** *A branching implementation re-*

lation R *is a binary relation on* $\delta \times \wp$ *such that if* $(M, I) \in R$ *then:*

1. $(\forall \ell, M')(M \xrightarrow{\ell}_{\mathrm{r}} M') \implies (\exists\, I_0, \ldots, I_n, I' \cdot$
   $I_i \xrightarrow{\tau} I_{i+1}\ \forall\, 0 \leq i < n\ \wedge\ I_n \xrightarrow{\hat{\ell}} I'\ \wedge$
   $I_0 = I\ \wedge\ (M, I_i) \in R\ \forall\, 0 \leq i \leq n\ \wedge$
   $(M', I') \in R)$
2. $(\forall \ell, I')(I \xrightarrow{\ell} I') \implies (\exists\, M_0, \ldots, M_n, M' \cdot$
   $M_i \xrightarrow{\tau}_{\mathrm{p}} M_{i+1}\ \forall\, 0 \leq i < n\ \wedge\ M_n \xrightarrow{\hat{\ell}}_{\mathrm{p}} M'\ \wedge$
   $M_0 = M\ \wedge\ (M_i, I) \in R\ \forall\, 0 \leq i \leq n\ \wedge$
   $(M', I') \in R)$

Let M *be an MTS, and* I *an LTS. We say that* I *is a branching implementation of* M*, written* $M \preceq_{\mathrm{b}} I$*, iff* $\alpha M = \alpha I$ *and there exists a branching implementation relation* R *such that* $(M, I) \in R$*.*

It can be clearly observed that if this relation is restricted to LTSs it coincides with branching equivalence. It can also be easily proved that if $M \preceq_{\mathrm{b}} I$ and $I \approx_{\mathrm{b}} I'$ then $M \preceq_{\mathrm{b}} I'$. Therefore, this novel implementation relation is a sound extension of branching equivalence.

In this way we have defined a new semantics over MTS that extends branching equivalence.

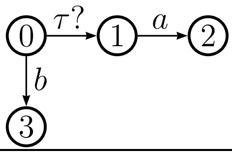
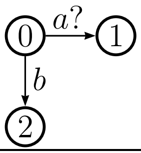
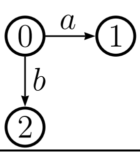
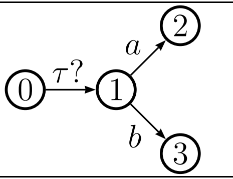
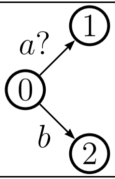
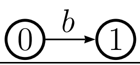
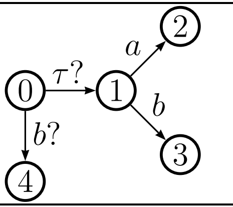
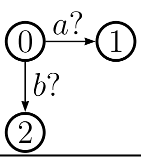
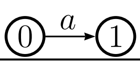
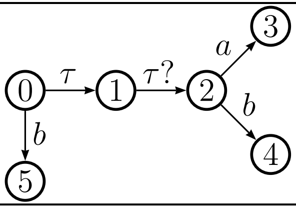
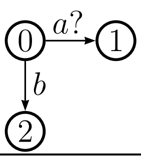
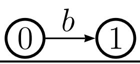
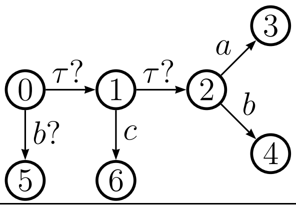
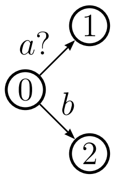
5.2.3 Validation

In Section 5.2.1 we explored the behaviour desired for a new MTS semantics based on the analysis of a series of examples. Furthermore, in section 5.2.2 we formally defined a new MTS semantics. In this section we validate that semantics by assessing if it complies with the expected behaviour for the examples analysed in Section 5.2.1 as well as for our running example. In order to do so, we should bear in mind that according to the definition of branching thorough semantics $\mathcal{N}$ is a refinement of $\mathcal{M}$ iff every implementation of $\mathcal{N}$ is also an implementation of $\mathcal{M}$. The strategy to validate the definition will be the following:

- Firstly, we examine the results achieved for all the examples for which intuitively we do not consider model $\mathcal{N}_i$ to be a refinement of model $\mathcal{M}_i$, i.e. $\mathcal{M}_i \preceq_{\mathrm{w}} \mathcal{N}_i$ but $\mathcal{M}_i \not\preceq_{\mathrm{b}} \mathcal{N}_i$. We demonstrate that the branching semantics matches the expected behaviour by showing a counterexample consisting of a model $\mathcal{I}_i$ that is actually an implementation of model $\mathcal{N}_i$ but not of $\mathcal{M}_i$ according to this newly defined semantics.
- Secondly, we study the examples we do consider to be valid refinements, i.e. $\mathcal{M}_i \preceq_{\mathrm{b}} \mathcal{N}_i$. Due to the infinite set of possible implementations, in order to validate that every implementation of $\mathcal{N}_i$ is also an implementation of $\mathcal{M}_i$, we analyse the results for at least one member of each of the equivalence classes of the set of implementations of $\mathcal{N}_i$ given by branching equivalence. To further complete the validation, we have also included for each example an LTS model that is not a valid branching implementation of $\mathcal{M}_i$ and therefore it should not be a valid branching implementation of $\mathcal{N}_i$ either.

All these tests have been performed using the software tool MTSA described in Chapter 7. The results are shown in the following tables, where True (False) in a cell indicates whether the model in that row is (is not) a branching implementation of the model in the corresponding column.

Table 5.1: Validating that $\mathcal{N}$ does not refine $\mathcal{M}$.

	$\mathcal{M}_1$: 	$\mathcal{N}_1$: 
$\mathcal{I}_{1_1}$: 	False	True
	$\mathcal{M}_2$: 	$\mathcal{N}_2$: 
$\mathcal{I}_{2_1}$: 	False	True
	$\mathcal{M}_3$: 	$\mathcal{N}_3$: 
$\mathcal{I}_{3_1}$: 	False	True
	$\mathcal{M}_4$: 	$\mathcal{N}_4$: 
$\mathcal{I}_{4_1}$: 	False	True
	$\mathcal{M}_6$: 	$\mathcal{N}_6$: 

$\mathcal{I}_{6_1}$: $$\begin{tikzcd} & 1 \\ 0 \arrow[ur, "a"] \arrow[dr, "b"'] & \\ & 2 \end{tikzcd}$$	False	True

Table 5.2: Validating that $\mathcal{N}$ refines $\mathcal{M}$.

	$\mathcal{M}_5$: $$\begin{tikzcd} & & & 3 \\ 0 \arrow[r, "\tau?"] & 1 \arrow[r, "\tau?"] & 2 \arrow[ur, "a"] \arrow[dr, "b"'] & \\ & & & 4 \end{tikzcd}$$	$\mathcal{N}_5$: $$\begin{tikzcd} & 1 \\ 0 \arrow[ur, "a"] \arrow[dr, "b"'] & \\ & 2 \end{tikzcd}$$
$\mathcal{I}_{5_1}$: $$\begin{tikzcd} & 1 \\ 0 \arrow[ur, "a"] \arrow[dr, "b"'] & \\ & 2 \end{tikzcd}$$	True	True
$\mathcal{I}_{5_2}$: $$\begin{tikzcd} & & 2 \\ 0 \arrow[r, "\tau"] & 1 \arrow[ur, "a"] \arrow[dr, "b"'] & \\ & & 3 \end{tikzcd}$$	True	True
$\mathcal{I}_{5_3}$: $$\begin{tikzcd} 0 \arrow[r, "a"] & 1 \end{tikzcd}$$	False	False

All the above results show that branching semantics matches the desired behaviour for a new MTS semantics. Furthermore, we have also validated this new semantics using the running example, testing if it rejects the undesired implementation depicted in Figure 5.1 as a valid implementation of the initial model shown in Figure 1.1. The result is that branching semantics rejects that implementation as we expected.

5.3 Branching Alphabet Semantics

5.3.1 Definition

Branching refinement, similarly to weak refinement, does not allow for the comparison of models with different alphabets. However, we can do so by using the hiding operator, i.e. hiding the new labels of the extended alphabet. For example, given a model M and a model N, the latter with an alphabet that extends the alphabet of M, i.e. $\alpha M \subseteq \alpha N$, in order to assess whether N is a refinement of M we compute $M \preceq N@\alpha M$.

This operation gives a new refinement, therefore defining a new semantics for MTSs for which it is possible to extend the alphabet of the models. We will now provide a formal definition for this novel semantics.

Definition 5.3.1 (Branching Alphabet Refinement)**.** *An MTS N is a* branching alphabet refinement *of an MTS M, written $M \preceq_{\text{ab}} N$, if $\alpha M \subseteq \alpha N$ and $M \preceq_{\text{b}} N@\alpha M$.*

Note that this new semantics is an extension of branching semantics, as they behave in the same way when comparing models with identical alphabets.

We now show that a sound relationship between branching implementation semantics and its alphabet extension exists, but first we define formally *equivalence* and *alphabet extension* for MTS.

Definition 5.3.2 (Equivalence)**.** *Given a refinement for MTS, $\preceq$, we say that M and N are* equivalent*, written $M \approx N$, iff $M \preceq N$ and $N \preceq M$. We shall sometimes subindex $\approx$ to explicitly note the underlying refinement relation, e.g. $\approx_{\mathrm{b}}$ for branching refinement $\preceq_{\mathrm{b}}$.*

Definition 5.3.3 (Alphabet Extension)**.** *Given an observational refinement for MTS, $\preceq$, we say that M' is an* alphabet extension *of M iff $M'@\alpha M \approx_{\mathrm{w}} M$.*

Theorem 5.3.4 (Branching semantics is sound w.r.t Alphabet Extension)**.** *Let M be an MTS and I be an LTS such that I is a branching implementation of M, i.e. $M \preceq_{\mathrm{b}} I$. Given M' an MTS that is a branching alphabet extension of M, then there exists I' a branching alphabet extension of I such that $M' \preceq_{\mathrm{b}} I'$.*

Proof. By theorem 5.3.9, M' and I are consistent since $\alpha M' \cap \alpha I = \alpha M$, M and I are consistent, $M'@\alpha M \preceq_{\mathrm{ab}} M$ and $I@\alpha M \preceq_{\mathrm{ab}} I$. Hence there exists I' a common implementation of M' and I, thus $I \preceq_{\mathrm{ab}} I'$ which is equivalent to $I \preceq_{\mathrm{b}} I'@\alpha I$. Considering that refinement restricted to LTSs coincides with branching bisumulation we obtain that $I \approx_{\mathrm{b}} I'@\alpha I$. □

Intuitively, if a model M is extended into a model M' then all implementations of M can be extended to be an implementation of M'. Figure 5.6 provides a graphical representation of this. We say, informally, that the diagram commutes, meaning that it is possible to obtain the same result by taking an implementation of M and then extending the alphabet of that implementation; or by extending the alphabet of M and then taking an implementation of that model.

From an engineering perspective this result implies that whatever implementation we have in mind for a given partial model, refining the alphabet of the partial

$$\begin{tikzcd}
M \arrow[r, "\begin{array}{c}\textit{branching}\\ \alpha\ \textit{extension}\end{array}"] \arrow[d, "\begin{array}{c}\textit{branching}\\ \textit{impl}\end{array}"'] \arrow[dr, dotted, "\textit{branching}\ \alpha\ \textit{impl}" sloped] & M' \arrow[d, "\begin{array}{c}\textit{branching}\\ \textit{impl}\end{array}"] \\
I \arrow[r, "\begin{array}{c}\textit{branching}\\ \alpha\ \textit{extension}\end{array}"'] & I'
\end{tikzcd}$$

Figure 5.6: Informally, alphabet extension and branching implementations commute.

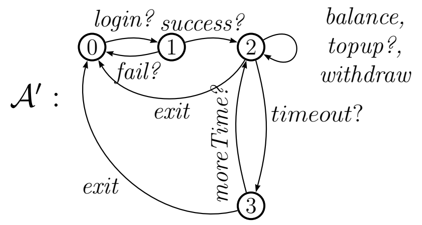


Figure 5.7: Extension of model $\mathcal{A}$ from Figure 1.1.

model will not rule out that implementation: extending the original implementation to make it an implementation of the new model is possible.

It is important to note that it is not possible to formulate a similar soundness result as the one above for weak semantics:

Remark 5.3.5 (Weak semantics is not sound w.r.t Alphabet Extension)**.** *Let M and M' be MTSs such that M' is a weak alphabet extension of M. It is not the case that for all LTSs I such that $M \preceq_{\mathrm{w}} I$ then there exists I' such that $M' \preceq_{\mathrm{w}} I'$ and I' is a weak alphabet extension of I.*

Proof.

Consider the example described in Section 5.1. Assume we extend model $\mathcal{A}$ given in Figure 1.1 to produce $\mathcal{A}'$ by extending its alphabet with the label *timeout*, and replacing the τ transition from state 2 to state 3 with a *timeout* transition as depicted in Figure 5.7. It would be reasonable to expect that model $\mathcal{K}$ (Figure 5.1) could be extended with *timeout* into a $\mathcal{K}'$ to obtain an implementation of $\mathcal{A}'$. However, this is not possible. If we analyse this in further detail, we can see that we would need $\mathcal{K}'$ to be able to perform a *timeout* after *success*. Hence, $\mathcal{K}'$ would have a new state in between *exit* and *timeout*. This leads to one of two options, either the new state does not simulate the required behaviour of state 2 of model $\mathcal{A}'$ because it does not have transitions *balance* and *withdraw*, and therefore $\mathcal{K}'$ could not be an implementation of $\mathcal{A}'$; or it does have those transitions and refines state 2 of model $\mathcal{A}'$, but in this case $\mathcal{K}'@\alpha\mathcal{K}$ would not be equivalent to $\mathcal{K}$ since $\mathcal{K}$ does not have any of the functionalities available after *success* and therefore $\mathcal{K}'$ could not be an alphabet extension of $\mathcal{K}$. $\square$

5.3.2 Consistency

In this section, we focus on analysing the notion of consistency under branching alphabet semantics and comparing the results with those applicable to weak alphabet semantics. In particular, we provide a complete characterization of consistency under branching alphabet semantics and show that, unlike in weak alphabet semantics, consistency is preserved by hiding non-shared actions.

The problem of characterising consistency has been solved for strong and weak semantics in Sections 3.3 and 4.2.2, where a sufficient and necessary condition for determining if there exists a common strong or weak refinement for two models is presented (similar results are unavailable for weak alphabet semantics). We now define a new relation, *branching alphabet consistency relation*, and show that it characterises branching alphabet consistency.

Definition 5.3.6 (Branching Alphabet Consistency Relation)**.** *A branching al-*

phabet consistency relation *is a binary relation* $C \subseteq \delta \times \delta$*, such that the following conditions hold for all* $(M, N) \in C$*:*

1. $(M \xrightarrow{\ell}_{\mathrm{r}} M') \implies (\exists\, N_0, \ldots, N_n, N') \cdot ((N_i \xrightarrow{v_i}_{\mathrm{p}} N_{i+1} \wedge v_i \notin \alpha M)\ \forall\, 0 \leq i < n \wedge$
$N_0 = N \wedge N_n \xrightarrow{\hat{\ell}}_{\mathrm{p}} N' \wedge (M, N_i) \in C\ \forall\, 0 \leq i \leq n \wedge (M', N') \in C$
2. $(N \xrightarrow{\ell}_{\mathrm{r}} N') \implies (\exists\, M_0, \ldots, M_n, M') \cdot ((M_i \xrightarrow{v_i}_{\mathrm{p}} M_{i+1} \wedge v_i \notin \alpha N)\ \forall\, 0 \leq i < n \wedge$
$M_0 = M \wedge M_n \xrightarrow{\hat{\ell}}_{\mathrm{p}} M' \wedge (M_i, N) \in C\ \forall\, 0 \leq i \leq n \wedge (M', N') \in C$

Intuitively, this relation requires that one model provides as possible behaviour at least all the required behaviour of the other, and vice versa.

The branching alphabet consistency relation defined above characterises branching alphabet consistency, as stated in the following theorem.

Theorem 5.3.7 (Characterisation of Branching Alphabet Consistency)**.** *MTSs* M *and* N *are branching alphabet consistent iff there exists a branching alphabet consistency relation* C_{MN} *such that* (M, N) *is in* C_{MN}*.*

Proof. $\Leftarrow$) Let CI be an LTS defined by $CI = (C_{MN}, \mathit{Act}, \Delta_{CI}, (M_0, N_0))$, where Δ_{CI} is the smallest relation that satisfies the following rules, assuming that $\{(M, N), (M', N') \subseteq C_{MN}\}$.

$$\frac{M \xrightarrow{\ell}_{\mathrm{r}} M',\ N \xrightarrow{\hat{\ell}}_{\mathrm{p}} N'}{(M,N) \xrightarrow{\ell} (M',N')} \qquad \frac{M \xrightarrow{\hat{\ell}}_{\mathrm{p}} M',\ N \xrightarrow{\ell}_{\mathrm{r}} N'}{(M,N) \xrightarrow{\ell} (M',N')} \qquad \frac{M \xrightarrow{\ell}_{\mathrm{p}} M',\ \ell \notin \alpha N}{(M,N) \xrightarrow{\ell} (M',N)} \qquad \frac{N \xrightarrow{\ell}_{\mathrm{p}} N',\ \ell \notin \alpha M}{(M,N) \xrightarrow{\ell} (M,N')}$$

It is easy to prove that $M \preceq CI$ using that

$$R = \{(M, (M, N)) \mid (M, N) \in C_{MN}\}$$

is a branching implementation relation between M and $CI@\alpha M$.

$\Rightarrow$) Since M and N are consistent we can take an LTS CI such that $M \preceq CI$, $N \preceq CI$ and $\alpha CI = \alpha M \cup \alpha N$. By definition of branching alphabet semantics there exist R_M and R_N implementation relations between M and $CI@\alpha M$, and between N and $CI@\alpha N$, respectively. Let C_{MN} be a relation defined by $C_{MN} = R_M \circ R_N^{-1}$. It can easily be proved that C_{MN} is a branching alphabet consistency relation between M and N. □

Note that the Branching Alphabet Consistency Relation is equivalent to branching bisimulation when restricted to LTSs with the same alphabet. This result is as expected, since an LTS is an MTS that characterises only one implementation, itself. Hence, it can only be consistent with any LTS that is equivalent to it; equivalence which in this case is that of LTS branching bisimulation.

In the same way Theorem 5.3.4 relates refinement with alphabet extension, it is interesting and relevant to analyse the relation between consistency and alphabet extension. Here we also find that the expected results hold for branching semantics but not for weak semantics.

The following theorem establishes that models are branching alphabet consistent if and only if they are branching consistent over their common alphabet.

Theorem 5.3.8. *Let M and N be MTSs, and $A = \alpha M \cap \alpha N$ be the common alphabet of M and N. $M@A$ and $N@A$ are branching consistent iff M and N are branching alphabet consistent.*

From an engineering point of view, this theorem expresses the fact that in order to assess whether two models are consistent it is sufficient to evaluate whether they are consistent over their common alphabet. On the other hand, it tells us that given two consistent models with the same alphabet it is possible to elaborate those models independently, extending their alphabets over different labels, knowing that the models will always remain consistent. This is a useful feature, especially when comparing two models taken from different viewpoints of the system, and for which there is a requirement to increase the level of detail with regards to different aspects. Interestingly, weak alphabet consistency does not satisfy the left-to-right implication of the above theorem. In other words, if two models are weak consistent, extending them over new labels does not guarantee they will remain consistent.

A related result, that in a way is more general than Theorem 5.3.8 is shown below. Note that the converse of Theorem 5.3.9 is not generally true, but in the particular case of Theorem 5.3.8 the converse is also true and it can be trivially proved.

Theorem 5.3.9. *Let M' and N' be MTSs, and $A = \alpha M' \cap \alpha N'$ be the common alphabet of M' and N'. If there exist MTSs M and N such that $M'@A \preceq_{\mathrm{ab}} M$, $N'@A \preceq_{\mathrm{ab}} N$, and M and N are branching alphabet consistent, then M' and N' are branching alphabet consistent.*

Proof. Since M and N are consistent there exists I a common implementation of them. Considering that $M'@A \preceq_{\mathrm{ab}} M$ and $N'@A \preceq_{\mathrm{ab}} N$ we get that I is a common implementation of $M'@A$ and $N'@A$. Therefore, by theorem 5.3.7 there exists a consistency relation, $C_{M'N'}$, between $M'@A$ and $N'@A$. Then, using that $M'@A$ and $N'@A$ have been produced by hiding the non-common alphabet of M' and N' respectively, it can be easily proved that $C_{M'N'}$ is a branching alphabet consistency relation between M' and N'. □

In summary, we have provided a complete characterization for consistency under branching alphabet semantics and shown that it has the expected properties when considered in the context of alphabet extension. These results do not exist for weak alphabet refinement of MTSs.

5.3.3 Merge

We will now introduce a $+_{cr}$ operator that given two branching alphabet consistent models produces a common refinement under this semantics. This operator follows the same pattern as with the other semantics, taking a conservative approach when putting together the common behaviour of the given models, resulting in a common refinement that might not be the LCR or an MCR even though these models may exist. However, unlike the same operator under weak alphabet semantics, given a pair of branching alphabet consistent models, the $+_{cr}$ can always produce a common refinement. This is due to the fact that a branching alphabet consistency relation can always be built for a pair of branching alphabet consistent models, while the $+_{cr}$ operator for weak alphabet semantics might fail because it cannot build a weak alphabet consistency relation for the given models.

Definition 5.3.10. (The $+_{cr}$ operator under Branching Alphabet Semantics) *Let $M = (S_M,\ A_M,\ \Delta^r_M,\ \Delta^p_M,\ s_{0M})$ and $N = (S_N,\ A_N,\ \Delta^r_N,\ \Delta^p_N,\ s_{0N})$ be MTSs and let C_{MN} be the largest branching alphabet consistency relation between them. $M +_{cr} N$ is the MTS $(C_{MN}, A_M \cup A_N, \Delta^r, \Delta^p, (s_{0M}, s_{0N}))$, where Δ^r and Δ^p are the smallest relations that satisfy the rules below, for $\ell \in Act_\tau$:*

$$\textbf{RP}\ \frac{M\xrightarrow{\hat{\ell}}_{\mathrm{r}}M',\ N\xrightarrow{\hat{\ell}}_{\mathrm{p}}N'}{(M,N)\xrightarrow{\ell}_{\mathrm{r}}(M',N')} \qquad \textbf{PR}\ \frac{M\xrightarrow{\hat{\ell}}_{\mathrm{p}}M',\ N\xrightarrow{\hat{\ell}}_{\mathrm{r}}N'}{(M,N)\xrightarrow{\ell}_{\mathrm{r}}(M',N')}$$

$$\textbf{PD}\ \frac{M\xrightarrow{\ell}_{\mathrm{p}}M',\ N\xrightarrow{\hat{\tau}}_{\mathrm{p}}N'}{(M,N)\xrightarrow{\ell}_{\mathrm{r}}(M',N')}\ \ell\notin(\alpha N\cup\{\tau\}) \qquad \textbf{DP}\ \frac{M\xrightarrow{\hat{\tau}}_{\mathrm{p}}M',\ N\xrightarrow{\ell}_{\mathrm{p}}N'}{(M,N)\xrightarrow{\ell}_{\mathrm{r}}(M',N')}\ \ell\notin(\alpha M\cup\{\tau\})$$

$$\textbf{PP}\ \frac{M\xrightarrow{\hat{\ell}}_{\mathrm{p}}M',N\xrightarrow{\hat{\ell}}_{\mathrm{p}}N'}{(M,N)\xrightarrow{\ell}_{\mathrm{p}}(M',N')}$$

In order to merge models under branching alphabet semantics we can parameterise Algorithm 4.2.17 using the branching alphabet refinement relation and the $+_{cr}$ operator for branching alphabet semantics, thus obtaining a merge algorithm for this semantics. The resulting algorithm is complete, i.e. if it is applied to two models that have an LCR or a set of MCRs it will find them. Therefore, it improves on the merge algorithm for weak alphabet semantics since it is not limited by the incompleteness of the underlying $+_{cr}$ operator for that semantics. As with the merge algorithm for strong or weak alphabet semantics, this algorithm is still limited by the intrinsic incompleteness of the merge operator. In this case, when two consistent models do not have an LCR nor a set of MCRs because there is an infinite sequence of common refinements where each model is more abstract than all the previous ones, the algorithm detects this situation and returns a common refinement.

Although it would be desirable that the result of merging two models was always well defined and unique, from an engineering point of view we can see the cases where the merge returns multiple MCRs or no MCRs exist as a chance to guide the modeller to elicit new requirements.

Based on our discussion on Section 3.4.4 we know that if the merge returns multiple MCRs, these models only differ in their initial states. In particular, we know that there are non-deterministic choices from the initial state and the models differ in which of those actions are required and which are maybe. Moreover, the different options cannot be reconciled into one MTS. This case can guide the modeller to further analyse which are the sources of non-determinism in the initial state and potentially gather more knowledge of the system that would allow him to select the most appropriate MCR.

We will now analyse the case where two models do not have a set of MCRs but infinite common refinements. In this case, each model has a loop with maybe and required transitions, and although the behaviour described by each of these loops is consistent with each other, it is not possible to reconcile their behaviour into a unique loop. This is because potentially on each iteration the decision about which must be required transitions and which are maybe transitions might alternate. This leads to infinite common refinements, where all these potential different options are unfolded and expressed. In this case the modeller can analyse how the repetitive behaviour of the two models should be combined and whether there is one alternative that is appropriate, eliminating the uncertainty that leads to the infinite different unfoldings.

Summarising, in this chapter we have defined a new observational semantics for MTSs that preserves the branching structure, thus avoiding the unintuitive implementations allowed by weak semantics. Furthermore, we have formally defined an extension of this semantics that supports not only the elaboration of model behaviour but also the extension of their alphabets, laying the foundations for a sound elaboration process where the level of detail of the models can be increased over time. We have also shown that extending the alphabet of a partial behaviour model is a sound operation with respect to branching semantics, while the same is not true for weak semantics. Finally, we have provided a characterisation of branching alphabet consistency and solved the problem of merge under this semantics. In Chapter 8 we will apply these results to a case study.

Chapter 6

Algebraic Properties of Merge

In practice, partial behaviour model construction, refinement and merging are likely to be combined in many different ways, possibly in conjunction with other operators on partial models, such as parallel composition. Therefore, it is essential to study their algebraic properties, to guarantee that the overall process yields sensible results. For example, does the order in which various partial models are merged matter? Is merging two models and elaborating the result through refinement equivalent to elaborating the models independently and then merging them? In this section, we aim to answer such questions. Specifically, we show that while the existence of multiple non-equivalent MCRs does not guarantee many of the properties that hold when LCRs exist, the right choice of an MCR among the possible options can be made in order to guarantee particular algebraic properties.

6.1 Properties of Parallel Composition[1]

We first study properties of the parallel composition operator proposed by Larsen et al. in [66, 52]. We study the relation between the implementations of two MTSs to be composed in parallel with the implementations of the model resulting from the application of the parallel composition operator. The results provide, on one hand, an insight into the semantics of the parallel composition operator, and on the other, property preservation results that are important to understand how merge and parallel composition can be used together.

Larsen and Thomsen [66] defined a parallel composition operator over MTSs, intended to describe how models of two different systems work together:

Definition 6.1.1. (Parallel Composition [66]) *Let $M = (S_M,\ A_M,\ \Delta^r_M,\ \Delta^p_M,\ s_{0M})$ and $N = (S_N,\ A_N,\ \Delta^r_N,\ \Delta^p_N,\ s_{0N})$ be MTSs.* Parallel composition *($\|$) is a symmetric operator such that $M\|N$ is the MTS $(S_M \times S_N,\ A_M \cup A_N,\ \Delta^r,\ \Delta^p,\ (s_{0M}, s_{0N}))$, where Δ^r and Δ^p are the smallest relations that satisfy the rules below, where $\ell \in Act_\tau$:*

$$\textbf{RD}\,\frac{M\xrightarrow{\ell}_{\mathrm{r}}M'}{M\|N\xrightarrow{\ell}_{\mathrm{r}}M'\|N}\;\ell\notin\alpha N \qquad \textbf{PR}\,\frac{M\xrightarrow{\ell}_{\mathrm{p}}M',\ N\xrightarrow{\ell}_{\mathrm{r}}N'}{M\|N\xrightarrow{\ell}_{\mathrm{p}}M'\|N'}\;\ell\neq\tau \qquad \textbf{PD}\,\frac{M\xrightarrow{\ell}_{\mathrm{p}}M'}{M\|N\xrightarrow{\ell}_{\mathrm{p}}M'\|N}\;\ell\notin\alpha N$$

$$\textbf{RR}\,\frac{M\xrightarrow{\ell}_{\mathrm{r}}M',N\xrightarrow{\ell}_{\mathrm{r}}N'}{M\|N\xrightarrow{\ell}_{\mathrm{r}}M'\|N'}\;\ell\neq\tau \qquad \textbf{PP}\,\frac{M\xrightarrow{\ell}_{\mathrm{p}}M',N\xrightarrow{\ell}_{\mathrm{p}}N'}{M\|N\xrightarrow{\ell}_{\mathrm{p}}M'\|N'}\;\ell\neq\tau$$

When restricted to LTSs, the parallel composition operator defined above becomes the standard one (e.g., [60]).

[1] I would like to thank Nicolas D'Ippolito for his contribution on the study of the properties of parallel composition of MTS. In particular, he has elaborated Theorems 6.1.4, 6.1.5 and 6.1.6, which are included in this section in order to provide a more comprehensive cover on the subject.

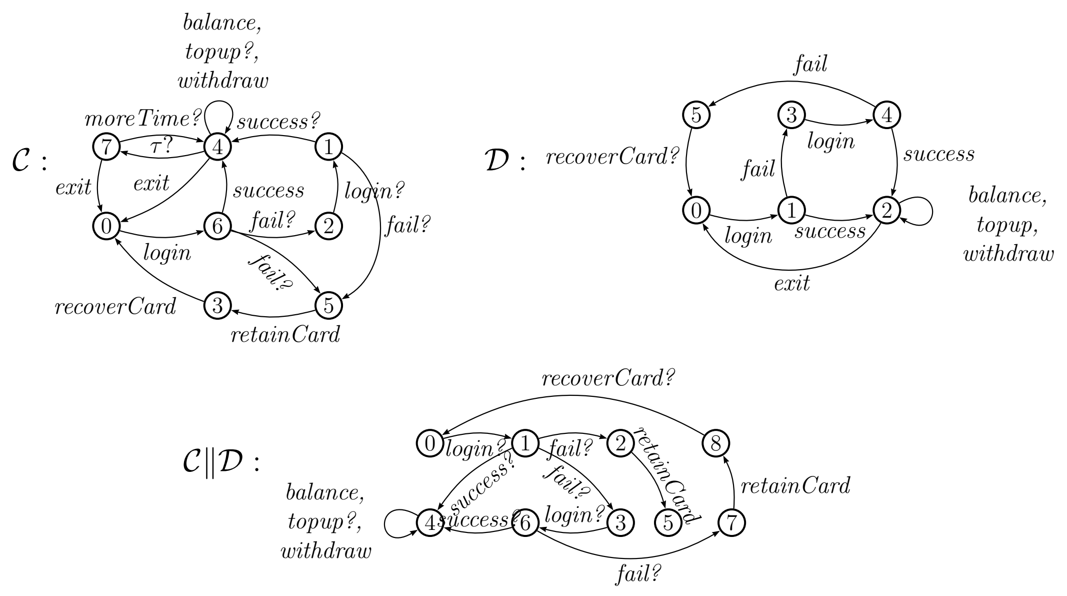


Figure 6.1: Model $\mathcal{C}$ from Figure 1.1, $\mathcal{D}$ models the assumptions on the user behaviour, and $\mathcal{C}\|\mathcal{D}$ is the parallel composition of these two models.

In the rules in Definition 6.1.1, "R" stands for "required", "P" stands for "possible", and "D" stands for "don't care". In particular, rule RR captures the case when there is a required transition in both models, PR — when there is a possible but not required transition in one model and a required transition in the other, and RD — when there is a required transition in one model on a non-shared action (i.e., on an action the other system is not concerned with).

Let's now apply the parallel composition operator to our running example. Figure 6.1 depicts a model characterising the ATM ($\mathcal{C}$), a model of the user ($\mathcal{D}$) and their parallel composition ($\mathcal{C}||\mathcal{D}$). This composition has a deadlock (see state 5 in model $\mathcal{C}||\mathcal{D}$) since composing the user model with an implementation of $\mathcal{C}$ that prohibits more than a single failed login can exhibit the following scenario: the user, after failing to login once, tries to login again (see state 3 in model $\mathcal{D}$) and yet the ATM does not allow it, instead attempting to retain the card (see state 5 in model $\mathcal{C}$). The two systems cannot synchronize, thus resulting in a deadlock.

We now recall some properties of parallel composition of MTS. Note that it does not preserve refinement. For instance, $\mathcal{C}\|\mathcal{D}$ is not a refinement of $\mathcal{C}$.

Property 6.1.2. *Parallel composition satisfies the following properties [52] (with $\alpha M = \alpha N$ and $\alpha P \subseteq \alpha M$):*

1. *(Commutativity)* $M\|N \quad = \quad N\|M$.
2. *(Associativity)* $(M\|N)\|P \quad = \quad M\|(N\|P)$.
3. *(Monotonicity)* $M \preceq N \quad \Rightarrow \quad M\|P \preceq N\|P$.

Before continuing with the study of the properties of parallel composition, we introduce the following definition of *deterministic* MTS which will be used for the remainder of this thesis. We say that an MTS is *deterministic* if it has no τ transitions and there is no state that has two outgoing possible transitions on the same label.

Definition 6.1.3. (Determinism) *Let $M = (S, A, \Delta^r, \Delta^p, s_0)$ be an MTS. M is* deterministic *iff $\tau \notin A$ and*

$$\forall s, s', s'' \in S \cdot (s \xrightarrow{\ell}_{\text{p}} s' \in \Delta^p \wedge s \xrightarrow{\ell}_{\text{p}} s'' \in \Delta^p) \implies (s' = s'').$$

We refer to the set of all deterministic implementations of an MTS M as $\mathcal{I}^{det}[M]$.

Composing two MTSs in parallel should result in a model that characterizes all pairwise parallel compositions of implementations of each of the MTSs. In other words, given MTSs M and N, it is expected that

$$\mathcal{I}[M||N] = \{I_M||I_N \mid I_M \in \mathcal{I}[M] \wedge I_N \in \mathcal{I}[N]\}) \qquad (6.1)$$

independently of the choice of refinement (strong, weak, weak alphabet, branching, branching alphabet).

However, this is not the case even under strong refinement. Consider the models in Figure 6.2. Model $I_{\mathsf{F}||\mathsf{G}}$ is a strong refinement of $\mathsf{F}||\mathsf{G}$. Yet it is easy to see that there are no implementations I_{F} and I_{G} of F and G, respectively, such that $I_{\mathsf{F}||\mathsf{G}} \equiv I_{\mathsf{F}}||I_{\mathsf{G}}$: In all implementations of F, if ℓ occurs, b is then enabled. In implementations of G, the trace ℓ, b must be possible. So the parallel composition of an implementation of F and G must either not have ℓ transitions, or it must allow the behaviour ℓ, b.

Although it is tempting to think that the problem is the non-deterministic choice in G, this is not the case. Consider models in Figure 6.3. Both H and I are deterministic, and $I_{\mathsf{H}||\mathsf{I}}$ is a strong refinement of $\mathsf{H}||\mathsf{I}$. Yet there are no I_{H} and I_{I} such that their parallel composition is equivalent to $I_{\mathsf{H}||\mathsf{I}}$. Intuitively, the problem is that if we pick implementations of H and I which admit b and a respectively, their parallel composition should admit any interleaving of these two actions. Yet in $I_{\mathsf{H}||\mathsf{I}}$, only one interleaving is allowed.

Summarizing, the MTS parallel composition operator in [52] produces a superset of the expected implementations (see Equation (6.1) above) independently of the choice of refinement:

Theorem 6.1.4. (Implementations of the MTS Parallel Composition Operator [31]) *For MTSs M and N,*

$$\mathcal{I}[M||N] \supseteq \{I_M||I_N \mid I_M \in \mathcal{I}[M] \wedge I_N \in \mathcal{I}[N]\}).$$

It is possible to enunciate restrictions that make the parallel composition operator

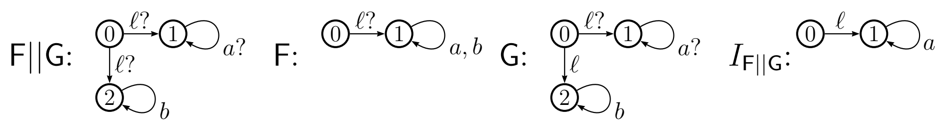


Figure 6.2: Examples for Parallel Composition: Non-Deterministic Models.

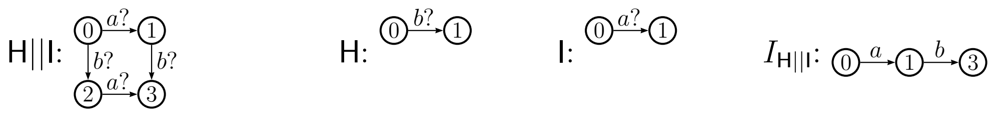


Figure 6.3: Example for Parallel Composition: Deterministic Models with Different Alphabets.

correct and complete with respect to a semantic definition along the lines of the one proposed in Equation (6.1). The restrictions are that the two MTSs to be composed in parallel have the same alphabet and that the operator yields a deterministic MTS. In addition, we must restrict the result to the universe of deterministic implementations:

Theorem 6.1.5. (Parallel Composition Preserves Deterministic Implementations [31]) *For MTSs M and N, if $\alpha M = \alpha N$ and $M||N$ is deterministic, then*

$$\mathcal{I}^{det}[M||N] = \{I_M||I_N \mid I_M \in \mathcal{I}^{det}[M] \wedge I_N \in \mathcal{I}^{det}[N]\})$$

under strong, weak, weak alphabet, branching and branching alphabet refinement.

Even though the parallel composition operator admits more implementations than it should (Theorem 6.1.4), the following result provides guarantees of property preservation and gives methodological guidelines as to how to use parallel composition in partial behaviour model elaboration.

The implementations characterized by $M\|N$ can be simulated by the parallel composition of *some choice* of implementations of M and N.

The notion of simulation between transition systems was originally introduced in [43]. A formal definition was presented in Chapter 2 (Definition 2.1.6).

Theorem 6.1.6. (Parallel Composition Preserves Simulation [31]) *Let M and N be MTSs and $I_{M||N}$ be an LTS. If $I_{M||N} \in \mathcal{I}[M||N]$, then*

$$\exists I_M \in \mathcal{I}[M], I_N \in \mathcal{I}[N] \;\cdot\; (\alpha I_M \cap \alpha I_N = \alpha M \cap \alpha N) \;\wedge\; (I_{M||N} \sqsubseteq_{\mathrm{s}} I_M \| I_N)$$

Given that simulation relations preserve safety properties ([1]), a corollary of the above theorem is that true safety properties are preserved by parallel composition. That is, if a safety property holds in an MTS, it also holds in its parallel composition with every other MTS.

The implications of the results discussed so far are that if, when elaborating the behaviour of the system-to-be, we have a partial description of the system and a partial behaviour of the environment, it is possible to reason compositionally about the safety properties of the composite system-environment. However, it is incorrect to compose these models in parallel and continue the elaboration process based on the composite model; elaboration must proceed in a component-wise fashion, refining the model of the system and of the environment separately. In fact, component-wise elaboration is standard for traditional approaches to behaviour modelling and analysis.

In Section 6.2, we show that the result on property preservation discussed above also plays a role in behaviour elaboration when using merge, more specifically, in the distributivity of merge over parallel composition.

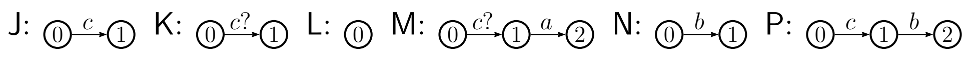


Figure 6.4: Example MTSs for algebraic properties.

6.2 Properties of LCRs

In this section, we discuss properties related to models for which the existence of a unique minimal common refinement can be guaranteed. In the next section, the uniqueness requirement is relaxed.

Property 6.2.1. *For MTSs M, N, and P, the following properties hold:*

1. *(Idempotence)* $\mathcal{LCR}_{M,M} \equiv M$.
2. *(Commutativity)* *If* $\exists \mathcal{LCR}_{M,N}$, *then* $\mathcal{LCR}_{M,N} \equiv \mathcal{LCR}_{N,M}$.
3. *(Associativity)* *If* $\exists \mathcal{LCR}_{M,N}$, $\exists \mathcal{LCR}_{P,\mathcal{LCR}_{M,N}}$, *and* $\exists \mathcal{LCR}_{N,P}$, *then* $\exists \mathcal{LCR}_{M,\mathcal{LCR}_{N,P}}$ *and* $\mathcal{LCR}_{P,\mathcal{LCR}_{M,N}} \equiv \mathcal{LCR}_{M,\mathcal{LCR}_{N,P}}$.

A useful property of LCRs is monotonicity with respect to refinement as it allows elaborating different viewpoints independently while ensuring that the properties of the original viewpoints put together still hold.

Property 6.2.2. (Monotonicity 1) *Let MTSs M, N, and P be given. If $\mathcal{LCR}_{M,N}$ exists, $M \preceq P$ and $N \preceq Q$, then $\mathcal{LCR}_{M,N} \preceq C$ for all $C \in \mathcal{CR}(P,Q)$.*

We now look at distributing merge over parallel composition: Assume that two stakeholders have developed partial models M and N of the intended behaviour of the same component. Each stakeholder will have verified that some required properties hold in a given context (other components and assumptions on the environment P_1, …, P_n). It would be desirable if merging viewpoints M and N preserved the properties of both stakeholders under the same assumptions on the environment, i.e., for $\mathcal{LCR}_{M,N} \parallel P_1 \parallel \cdots \parallel P_n$.

The following property supports the above reasoning:

Property 6.2.3. (Monotonicity 2) *If* $M \preceq N$ *and* $\alpha P \subseteq \alpha M$*, then* $M \| P \preceq N \| P$*.*

6.3 Properties of MCRs

In this section, we present algebraic properties of merging without assuming the existence of the LCR. The algebraic properties are therefore stated in terms of sets and the different choices that can be made when picking an MCR. Idempotence is the only property of Section 6.2 that still holds as is, since an LCR always exists between a system and itself. The rest of the properties discussed in Section 6.2 require some form of weakening.

Commutativity of merge holds independently of the existence of an LCR. The following property states that the set of MCRs obtained from M and N is the same as those obtained from N and M.

Property 6.3.1. (Commutativity) $\mathcal{MCR}(M, N) = \mathcal{MCR}(N, M)$*.*

On the other hand, associativity cannot be guaranteed the same way as commutativity. That is, it cannot be guaranteed that the same MCRs are achieved regardless of the order in which the three MTSs are merged. However, *the set of implementations* reachable through refinement is not affected by the merge order.

Property 6.3.2. (Associativity) *Let* $\mathcal{I}(X) = \bigcup_{x \in X} \mathcal{I}(x)$ *and let* M*,* N*, and* P *be MTSs. Then,*

$$\mathcal{I}\Big(\bigcup_{A \in \mathcal{MCR}(N,P)} \mathcal{MCR}(M,A)\Big) = \mathcal{I}\Big(\bigcup_{A \in \mathcal{MCR}(M,N)} \mathcal{MCR}(A,P)\Big).$$

From a practical perspective, the above property says that an engineer with a specific implementation in mind is able to reach it through successive refinements, regardless of the merge order of the three models. However, if the goal is not to achieve a specific implementation but rather obtain a particular partial model characterizing the implementations that conform to the three MTSs, then the merge order becomes important. This problem can be solved by defining an n-ary merge, as discussed in Section 9.3.

Monotonicity is also disrupted by multiple MCRs. It is not expected that any choice from $\mathcal{MCR}(M,N)$ is refined by any choice from $\mathcal{MCR}(P,N)$ when M is refined by P, because incompatible decisions may be made in the two merges. Rather, there are two desirable forms of monotonicity: (1) whenever a choice from $\mathcal{MCR}(M,N)$ is made, a choice from $\mathcal{MCR}(P,N)$ can be made such that a refinement holds; and (2) whenever a choice from $\mathcal{MCR}(P,N)$ is made, some model in $\mathcal{MCR}(M,N)$ can be chosen for a refinement to exist.

Form (1) does not hold, as the following example shows. Consider models K and N in Figure 6.4 with $\alpha\mathsf{K} = \{\mathtt{c}\}$ and $\alpha\mathsf{N} = \{\mathtt{b}\}$. These models are consistent, and their merge may result in model $\mathsf{P} \in \mathcal{MCR}(\mathsf{K},\mathsf{N})$. Also, $\mathsf{K} \preceq \mathsf{L}$ (assuming that $\alpha\mathsf{L} = \{\mathtt{c}\}$) and models L and N are consistent. However, $\mathcal{LCR}_{\mathsf{L},\mathsf{N}}$ is equivalent to N over $\{\mathtt{b},\mathtt{c}\}$, and since $\mathsf{N} \not\preceq \mathsf{P}$, no MCR of L and N that refines P can be chosen.

Form (1) fails because there are two choices of refinement being made. On the one hand, by picking one minimal common refinement for M and N over others, we are choosing one of several incompatible refinements. On the other hand, we are also choosing how to refine M into P. These two choices might be inconsistent, leading to the failure of monotonicity. This tells us that choosing an MCR adds information to the merged model, which may be inconsistent with evolutions of the different viewpoints that are represented by the models being merged.

Form (2) always holds, as stated below.

Property 6.3.3. (Monotonicity) *If M, N, P, and Q are MTSs, then:*

$$M \preceq P \wedge N \preceq Q \Rightarrow \forall B \in \mathcal{MCR}(P, Q) \cdot \exists A \in \mathcal{MCR}(M, N) \cdot A \preceq B.$$

Thus, once a model in $\mathcal{MCR}(P, Q)$ is chosen, there always exists some model in $\mathcal{MCR}(M, N)$ that it refines, and so the properties of each MCR of M and N are preserved by the MCRs of P and Q. If $\mathcal{MCR}(M, N)$ is a singleton set, Property 6.3.3 reduces to Property 6.2.3, as expected. In practical terms, this means that if the various viewpoints are still to be elaborated, the results of reasoning about one of their possible merges (picked arbitrarily) are not guaranteed to carry through once the viewpoints have been further refined.

In this section, we have shown that properties which hold for LCRs do not hold when consistent models have no unique MCR. Intuitively, the existence of non-equivalent MCRs implies that merging involves a choice that requires some form of human intervention: a choice which requires domain knowledge. While this affects some of the algebraic properties of merge, we have shown that these properties do hold in terms of preservation of implementations.

6.4 Proofs

In this section, we give proofs and proof sketches for various theorems and properties presented in this chapter.

Before proving Theorem 6.1.4, we introduce the following lemma.

Lemma 6.4.1. (Equivalence LTS-MTS Parallel Composition) *The MTS parallel composition operator restricted to LTSs is equivalent to the LTS parallel composition operator.*

Proof. Restricting a given MTS M to its required behaviour yields the exact same rules as shown in Definition 6.1.1. □

Theorem 6.4.2 (6.1.4)**.** (Implementations of the MTS Parallel Composition Operator [31]) *For MTSs M and N,*

$$\mathcal{I}[M||N] \supseteq \{I_M||I_N \mid I_M \in \mathcal{I}[M] \wedge I_N \in \mathcal{I}[N]\}).$$

Proof. By Lemma 6.4.1 and Proposition 4.1 in [66]. □

The following definition will be used below.

Definition 6.4.3. (Possible LTS) *Let $M = (S, A, \Delta^r, \Delta^p, s_0)$ be an MTS. We define $M_p = (S, A, \Delta^p, s_0)$ as the* possible LTS *of M.*

Before proving the next theorem, we introduce a lemma with the following intuition. Given an LTS I and an MTS M, if the behaviour of I over the alphabet of M can be simulated by the possible behaviour of M (as per Definition 6.4.3, we refer to it as M_p), through a relation T (written $I@\ \alpha M \sqsubseteq^{\mathrm{T}} M_p$), then it is possible to extend $R = T^{-1}$ and the behaviour of I, into R' and I' such that M is refined by I' over the alphabet of M through R' (written $M \preceq^{R'} I'@\ \alpha M$).

Lemma 6.4.4. (Simulation Relation Extension) *Let M be an MTS, I be an LTS, T be a relation such that $T \subseteq S_I \times S_M$ and $R = T^{-1}$. If $I@\ \alpha M \sqsubseteq^{\mathrm{T}} M_p$, then it is possible to extend R and the behaviour of I into R' and I', respectively, such that $M \preceq^{R'} I'@\ \alpha M$.*

Proof. The proof of this lemma is constructive, using Algorithm 6.4.5 executed with M, I and T, s.t. $I \sqsubseteq^{\mathrm{T}} M_p$, as inputs. With these inputs, provided that $I@\ \alpha M \sqsubseteq^{\mathrm{T}} M_p$, the algorithm extends $R = T^{-1}$ and the behaviour of I into R' and I' such that $M \preceq^{R'} I'@\ \alpha M$.

The algorithm first initializes the variables $I' = I$, $R' = R = T^{-1}$, $toVisit = \emptyset$ and $visited = \emptyset$. Then it loops, adding the required behaviour of M that is needed in I' and R' to satisfy both refinement conditions.

In every step, the set of visited pairs grows by one, and the number of pairs to visit is bounded (by $(|S_I|^3 \times |S_M|)$); thus, the algorithm terminates. Moreover, in each step R'^{-1} is preserved as a simulation relation, and the required behaviour is added only if it is needed, which means that R' satisfies both refinement conditions *in one step* and therefore, is "closer" to being a refinement relation.

Finally, when the procedure terminates, both refinement conditions hold, and therefore, $M \preceq^{R'} I'@\ \alpha M$. □

Algorithm 6.4.5. *Simulation Relation Extension*

Procedure Relation Extension (Relation T, MTS M, LTS I)

```
$R \leftarrow T^{-1}$; $R' \leftarrow R$; $I' \leftarrow I$
$toVisit \leftarrow \{(s_M, s_{I'}) | (s_M, s_{I'}) \in R'\}$; $visited \leftarrow \emptyset$
  **For all** $(s_M, s_{I'}) \in toVisit$
    $visited \leftarrow visited \cup \{(s_M, s_{I'})\}$
    **For all** $s_M \xrightarrow{\ell}_{\mathrm{r}} s'_M \cdot s_M \xrightarrow{\ell}_{\mathrm{r}} s'_M \in \Delta^r_M$
      **For all** $s'_{I'} \in Closure(s_{I'}, I')$
        **If** $\exists s''_{I'} \in S_{I'} \cdot s'_{I'} \xrightarrow{\ell} s''_{I'} \in \Delta_{I'}$
          **Let** $s'''$ : $state$
          **For all** $s''_{I'} \in S_{I'} \cdot s'_{I'} \xrightarrow{\ell} s''_{I'} \in \Delta_{I'}$
            **If** $(s'_M, s''_{I'}) \notin R'$
              // $s_M$ simulates $s'_I$ by $R'$, which means that there should
              //  be a transition (possibly including $\tau$-transitions) over
              // $\ell$ from $s_M$ to $s''_M$ (with $s_M \neq s''_M$), and the pair
              // $(s''_M, s''_{I'})$ is in $R'$. Therefore, $s_M \xrightarrow{\ell}_{\mathrm{r}} s'_M$ must be a
              // "non-deterministic" transition.
              $S_{I'} \leftarrow S_{I'} \cup \{s'''\}$
              $\Delta_{I'} \leftarrow \Delta_{I'} \cup \{s_{I'} \xrightarrow{\ell} s''_{I'}\}$
              $R' \leftarrow R' \cup \{(s'_M, s''')\}$
              **If** $(s'_M, s''') \notin visited$
                $toVisit \leftarrow toVisit \cup \{(s'_M, s''')\}$
            **EndIf**
          **EndFor**
        **Else**
          // We found a required transition which is not part of the
          // behaviour of $I'$.
          **If** $\exists s''_{I'} \in S_{I'} \cdot (s'_M, s''_{I'}) \in R'$
            // In this case, $(s'_M, s''_{I'})$ were already in $R$, which means that
            // we need to add another transition between them.
            $\Delta_{I'} \leftarrow \Delta_{I'} \cup \{s_{I'} \xrightarrow{\ell} s''_{I'}\}$
          **Else**
            **Let** $s'''$ : $state$
            $S_{I'} \leftarrow S_{I'} \cup \{s'''\}$
            $\Delta_{I'} \leftarrow \Delta_{I'} \cup \{s_{I'} \xrightarrow{\ell} s'''\}$
            $R' \leftarrow R' \cup \{(s'_M, s''')\}$
            **If** $(s'_M, s''') \notin visited$
              $toVisit \leftarrow toVisit \cup \{(s'_M, s''')\}$
          **EndIf**
        **EndIf**
      **EndFor**
    **EndFor**
  $toVisit \leftarrow toVisit \setminus \{(s_M, s_{I'})\}$
**EndFor**
**Return** $(R', I')$
```

Theorem 6.4.6 (6.1.5)**.** (Parallel Composition Preserves Deterministic Implementations [31]) *For MTSs M and N, if $\alpha M = \alpha N$ and $M||N$ is deterministic, then*

$$\mathcal{I}^{det}[M||N] = \{I_M||I_N \mid I_M \in \mathcal{I}^{det}[M] \wedge I_N \in \mathcal{I}^{det}[N]\})$$

under strong, weak, weak alphabet, branching and branching alphabet refinement.

Proof. ($\Leftarrow$) By Theorem 6.1.4.

($\Rightarrow$)

i) We construct I_M and I_N by the same process used in the proof of Theorem 6.1.6. It is easy to see that bisimulation conditions between I and $I_M||I_N$ hold if the behaviour of $I_M||I_N$ is restricted to the subset of transitions to and from states of the form $(s_{I_{M||N}}, s_{I_{M||N}})$.

ii) Now we show that every transition of $I_M||I_N$ *is* of the form $(s_{I_{M||N}}, s_{I_{M||N}})$, i.e.,

$$\forall (s_{I_{M||N}}, s_{I_{M||N}}) \in S_{I_M||I_N} \;\cdot\; (s_{I_{M||N}}, s_{I_{M||N}}) \xrightarrow{\ell} (s'_{I_{M||N}}, s'_{I_{M||N}}).$$

Let R, R_M and R_N be relations such that $M||N \preceq^{R} I_{M||N}$, $M \preceq^{R_M} I_M$, $N \preceq^{R_N} I_N$, respectively. By *i*), $(s_{0_{I_{M||N}}}, s_{0_{I_{M||N}}}) \in S_{I_M||I_N}$ holds.

We proceed by proof by contradiction. Let's choose a state $(s_{I_{M||N}}, s_{I_{M||N}})$ such that our hypothesis does not hold, i.e.,

$$\exists (i,j) \in S_{I_M||I_N} \cdot (s_{I_{M||N}}, s_{I_{M||N}}) \xrightarrow{\ell} (i,j) \in \Delta_{I_M||I_N} \wedge i \neq j$$

Then,

$$
\begin{aligned}
& (M||N \text{ and } I_{M||N} \text{ satisfy Definition 6.1.3, } \alpha M = \alpha N) \\
\Rightarrow\ & \nexists s'_{I_{M||N}} \cdot s_{I_{M||N}} \xrightarrow{\ell} s'_{I_{M||N}} \in \Delta_{I_{M||N}} \\
& (\text{Definition 6.1.1}) \\
\Rightarrow\ & (i \in S_M \cdot s_{I_M} \xrightarrow{\ell} i \in \Delta_{I_M}) \ \wedge \ (j \in S_N \cdot s_{I_N} \xrightarrow{\ell} j \in \Delta_{I_N}) \\
& (\text{Procedure in proof of Lemma 6.4.4}) \\
\Rightarrow\ & (\exists s'_M \in S_M \cdot (s'_M, i) \in R_M \wedge s_M \xrightarrow{\ell}_{\mathrm{r}} s'_M \in \Delta^r_M) \wedge \\
& (\exists s'_N \in S_N \cdot (s'_N, j) \in R_N \wedge s_N \xrightarrow{\ell}_{\mathrm{r}} s'_N \in \Delta^r_N) \\
& (\text{Definition 6.1.1}) \\
\Rightarrow\ & (s_M, s_N) \xrightarrow{\ell}_{\mathrm{r}} (s'_M, s'_N) \in \Delta^r_{M||N} \\
& (\text{Definition 3.1.1}) \\
\Rightarrow\ & \exists s \in S_{I_{M||N}} \cdot ((s'_M, s'_N), s) \in R \wedge s_{I_{M||N}} \xrightarrow{\ell} s \in \Delta_{I_{M||N}} \\
& (\text{Our assumption}) \\
\Rightarrow\ & \mathbf{false}
\end{aligned}
$$

Therefore, every state in $I_M||I_N$ is of the form $(s_{I_{M||N}}, s_{I_{M||N}})$.

By *i)* and *ii)*, we showed that $I \sim I_M||I_N$ and thus the theorem holds. □

Before stating and proving Theorem 6.1.6, we introduce another lemma. Intuitively, it states that given two MTSs, M and N, and an LTS I, if $M||N$ is refined by I over the alphabet of $M||N$, then I can be simulated (over the alphabet of M) by M.

Lemma 6.4.7. (Decomposition Simulation) *Let M and N be MTSs and I be an*

LTS. If $M||N \preceq I@(\alpha M \cup \alpha N)$, then the following holds:

$$I@\alpha M \ \sqsubseteq_{\mathrm{s}} M_p$$

Proof. We first define a relation S as follows:

$$S = \{(s_I, s_M) | \exists s_N \cdot ((s_M, s_N), s_I) \in R\}$$

We now show by contradiction that S is a simulation relation. Intuitively, if $I@\alpha M \ \sqsubseteq_{\mathrm{s}} M_p$ does not hold, then the MTS refinement condition 2 on $M||N \preceq^R I@(\alpha M \cup \alpha N)$ does not hold either. We check the conditions for simulation:

(1) By definition of S, $(s_{0_I}, s_{0_M}) \in S$

(2) By definition of S, $(s'_I, s'_M) \in S$. Assume that $\neg\exists s'_M \cdot s_M \overset{\ell}{\Longrightarrow}_{\mathrm{p}} s'_M$.

$$\begin{aligned} & \text{(Definition 6.1.1)} \\ \Rightarrow\ & s_N \overset{\ell}{\Longrightarrow}_{\mathrm{p}} s'_N \in \Delta^p_N \wedge \ell \in (\alpha N \setminus \alpha M) \cup \tau \\ & \text{(Definition 4.1.1)} \\ \Rightarrow\ & s_M (\overset{\ell}{\Longrightarrow}_{\mathrm{p}})^0 s_M \wedge (s'_I, s_M) \in S \end{aligned}$$

This contradicts the assumption. Thus, simulation condition 2 holds. □

Theorem 6.4.8 (6.1.6)**.** (Parallel Composition Preserves Simulation [31]) *Let M and N be MTSs and $I_{M||N}$ be an LTS. If $I_{M||N} \in \mathcal{I}[M||N]$, then*

$$\exists I_M \in \mathcal{I}[M], I_N \in \mathcal{I}[N] \ \cdot \ (\alpha I_M \cap \alpha I_N = \alpha M \cap \alpha N) \ \wedge \ (I_{M||N} \sqsubseteq_{\mathrm{s}} I_M \| I_N)$$

Proof. By Lemma 6.4.7, $I_{M||N}@\,\alpha M \sqsubseteq_{\mathrm{s}} M_p$ and $I_{M||N}@\,\alpha N \sqsubseteq_{\mathrm{s}} N_p$; by Lemma 6.4.4, it is possible to build I_M and I_N such that $I_M \in \mathcal{I}[M]$ and $I_N \in \mathcal{I}[N]$ by applying Algorithm 6.4.5.

We now define Q as follows:

$$Q = \{(s_{I_{M||N}}, (s_{I_{M||N}}, s_{I_{M||N}}))|s_{I_{M||N}} \in S_{I_{M||N}}\}$$

By construction of I_M and I_N, $I_{M||N} \sqsubseteq^{\mathrm{Q}} I_M||I_N$ and $\alpha I_M \cap \alpha I_N = \alpha M \cap \alpha N$. □

Lemma 6.4.9. (Alphabet reduction) *If A and B are sets and M and N are MTSs, then*

$$B \subseteq A \Rightarrow (M@A \preceq N@A \Rightarrow M@B \preceq N@B).$$

Proof. We show that the refinement relation that exists because $M@A \preceq_w N@A$ is also a refinement relation between $M@B$ and $N@B$, hence leading to $M@B \preceq_w N@B$. □

Property 6.4.10 (6.2.1)**.** *For MTSs M, N, and P, the following properties hold:*

1. *(Idempotence)* $\mathcal{LCR}_{M,M} \equiv M$.
2. *(Commutativity)* *If* $\exists\mathcal{LCR}_{M,N}$, *then* $\mathcal{LCR}_{M,N} \equiv \mathcal{LCR}_{N,M}$.
3. *(Associativity)* *If* $\exists\mathcal{LCR}_{M,N}$, $\exists\mathcal{LCR}_{P,\mathcal{LCR}_{M,N}}$, *and* $\exists\mathcal{LCR}_{N,P}$, *then* $\exists\mathcal{LCR}_{M,\mathcal{LCR}_{N,P}}$ *and* $\mathcal{LCR}_{P,\mathcal{LCR}_{M,N}} \equiv \mathcal{LCR}_{M,\mathcal{LCR}_{N,P}}$.

Proof. (1) and (2) follow straightforwardly from the definition of LCR (see Definition 3.4.2). (3) Let Q be $\mathcal{LCR}_{P,\mathcal{LCR}_{M,N}}$.

$$\begin{array}{ll}
& \mathcal{LCR}_{M,\mathcal{LCR}_{N,P}} \preceq Q \\
& \text{(Definition 3.4.2)} \\
& \mathcal{LCR}_{N,P} \preceq Q@(\alpha N \cup \alpha P) \wedge M \preceq Q@\alpha M \\
& \text{(Definition 3.4.2)} \\
\Leftarrow & Q@(\alpha N \cup \alpha P) \in \mathcal{CR}(N,P) \wedge M \preceq Q@\alpha M \\
& \text{(Definition 3.4.1)} \\
\Leftarrow & N \preceq Q@\alpha N \wedge M \preceq Q@\alpha M \wedge P \preceq Q@\alpha P \\
& \\
& (N \preceq \mathcal{LCR}_{M,N}@\alpha N \preceq Q@\alpha N \wedge M \preceq \mathcal{LCR}_{M,N}@\alpha M \preceq Q@\alpha M) \\
& \text{(Lemma 6.4.9)} \\
\Leftarrow & N \preceq \mathcal{LCR}_{M,N}@\alpha N \wedge M \preceq \mathcal{LCR}_{M,N}@\alpha M \wedge \mathcal{LCR}_{M,N} \preceq Q@(\alpha M \cup \alpha N) \wedge \\
& \wedge P \preceq Q@\alpha P \\
& \text{(Definition 4.2.2 and Definition 3.4.2)} \\
= & \mathbf{t}.
\end{array}$$

The other direction ($Q \preceq \mathcal{LCR}_{M,\mathcal{LCR}_{N,P}}$) is proven similarly. □

Property 6.4.11 (6.2.2)**.** (Monotonicity 1) *Let MTSs M, N, and P be given. If $\mathcal{LCR}_{M,N}$ exists, $M \preceq P$ and $N \preceq Q$, then $\mathcal{LCR}_{M,N} \preceq C$ for all $C \in \mathcal{MCR}(P,Q)$.*

Proof. Let $C \in \mathcal{MCR}(P, Q)$.

$$\begin{aligned}
& \text{(Since } P \preceq C@\alpha P,\ Q \preceq C@\alpha Q,\ \alpha P = \alpha M, \text{ and } \alpha Q = \alpha N) \\
\Rightarrow\ & (M \preceq P \Rightarrow M \preceq C@\alpha M) \wedge (N \preceq Q \Rightarrow N \preceq C@\alpha N) \\
& \text{(Properties of } \Rightarrow \text{ and } \wedge\text{, and Definition 3.4.1)} \\
\Rightarrow\ & (M \preceq P) \wedge (N \preceq Q) \Rightarrow C \in \mathcal{CR}(M, N) \\
& \text{(Definition 3.4.2)} \\
\Rightarrow\ & (M \preceq P) \wedge (N \preceq Q) \Rightarrow \mathcal{LCR}_{M,N} \preceq C
\end{aligned}$$

□

Property 6.4.12 (6.2.3)**.** (Monotonicity 2) *If $M \preceq_a N$ and $\alpha P \subseteq \alpha M$, then $M \| P \preceq_a N \| P$.*

Proof. $M \preceq_{\mathrm{wa}} N$ implies $M \preceq_{\mathrm{w}} N@\alpha M$. From [52], we then have $M \| P \preceq_{\mathrm{w}} (N@\alpha M \| P)$. Given that $\alpha P \subseteq \alpha M$, we have $(N@\alpha M \| P) = (N \| P)@\alpha M$. Hence $M \| P \preceq_{\mathrm{w}} (N \| P)@\alpha M$ which by definition means that $M \| P \preceq_{\mathrm{wa}} (N \| P)$. □

Property 6.4.13 (6.3.3)**.** (Monotonicity) *If M, N, P, and Q are MTSs, then:*

$$M \preceq P \wedge N \preceq Q \Rightarrow \forall B \in \mathcal{MCR}(P, Q) \cdot \exists A \in \mathcal{MCR}(M, N) \cdot A \preceq B$$

Proof. Assume $B \in \mathcal{MCR}(P, Q)$. Then $N \preceq Q \preceq B \wedge M \preceq P \preceq B$ which entails $B \in \mathcal{CR}(M, N)$. Hence either $B \in \mathcal{MCR}(M, N)$ or $\exists A \in \mathcal{MCR}(M, N) \cdot A \preceq B$. □

Chapter 7

Tool Support

As part of this thesis we have developed a tool, the Modal Transition System Analyzer (MTSA), which builds upon the Labelled Transition System Analyzer (LTSA) [60], extending it to support the construction and analysis of MTS models. In MTSA we have implemented the algorithms for computing refinement, consistency and merge for the different semantics analysed in this thesis. The basic mechanism for describing MTS models is using a text language based on the FSP process algebra [60], and includes operators such as sequential and parallel composition, and hiding, in addition to the MTS merge operator. The tool also supports visualization of MTSs in a graphical format, analyses such as animation, consistency checking, as well as deadlock freedom and refinement checks.

MTSA is available as an Open Source project in order to allow other researchers to contribute or to use the code base to write their own tools (available at `http://sourceforge.net/projects/mtsa/`).

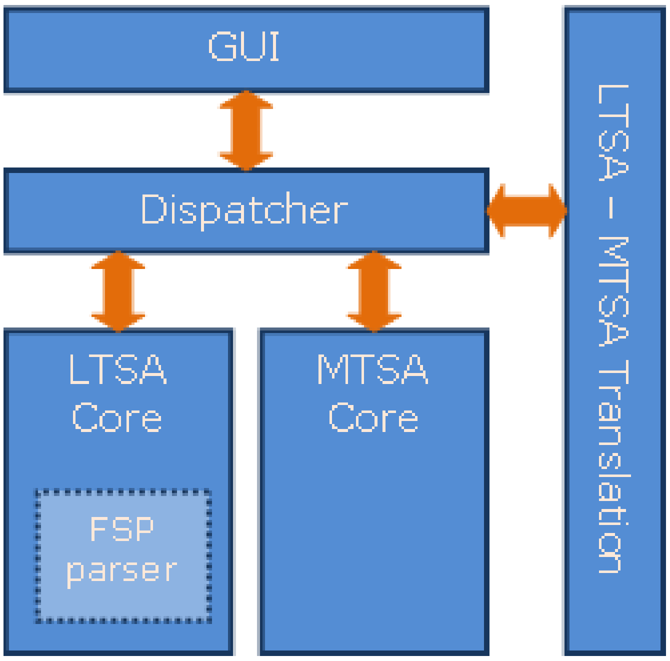


Figure 7.1: Architecture diagram of the MTSA tool.

Figure 7.1 shows a diagram of the architecture of MTSA. The system is divided in five main components organized in three layers.

The top layer is the *Graphical User Interface* (GUI), which controls the interaction with the user and implements the different input/output panels. This component is originally from LTSA and it was extended with new functionality specific for MTSs.

The second layer is the *Dispatcher*, which serves as an abstraction between the GUI and the *Core*.

The Core is where the actual algorithms and data structures for LTSs and MTSs are implemented. In this layer there are two main components, the *LTSA Core* and the *MTSA Core*. The LTSA Core comes from the original implementation of LTSA, and it includes the FSP parser and other algorithms for LTSs such as model checking and simulation. This component also provides the data structure to represent an LTS, which is used by all the algorithms of the LTSA Core as well as by the GUI. The FSP parser was extended in order to support the new operators defined for MTSs. The MTSA Core is a new module implemented to support all the different algorithms for MTSs presented in this thesis. This module is completely decoupled from the rest of the system and can easily be embedded into other systems that require any of the algorithms or data structures provided as part of this module.

In order to be able to reuse and leverage the GUI and the FSP parser from the LTSA code base we have adopted the following encoding of an MTS into a an LTS. Given an MTS $M = (S, L, \Delta^r, \Delta^p, s_0)$ we encode M as an LTS $M' = (S, L', \Delta, s_0)$ where

$$L' = L \cup \{\ell? \mid \ell \in L\}$$

and

$$\Delta = \Delta^r \cup \{s \xrightarrow{\ell?} s' \mid s \xrightarrow{\ell} s' \in \Delta^p \wedge s \xrightarrow{\ell} s' \notin \Delta^r\}.$$

In other words, we encode an MTS M into an LTS M' by introducing two labels into M' for each label in M, one being exactly the original label and the other one with a question mark added at the end. For each required transition on M we have a transition on M' using the original label, and for each maybe transition on M we have a transition on M' using the corresponding label with a question mark added at the end. This transformation is implemented in the *LTSA - MTSA Translation* component and it is used by the Dispatcher in order to adapt the corresponding data structures when invoking functions from the different modules.

In the rest of this chapter we will first give a brief introduction to FSP followed by a more detailed description of MTSA, including examples of how to apply the different algorithms.

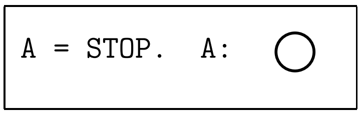
7.1 Finite State Processes

Finite State Processes (FSP) is a process algebra notation with a semantics in terms of LTS introduced by Magee et al [60]. It has been designed to textually specify LTS in a concise way.

A process written in FSP is given by an expression consisting of composition operators, processes and actions. While processes' names begin with an uppercase letter, actions' names start with a lowercase letter. A process is defined by one or more local processes separated by commas, and the end of the definition is marked with a full stop.

We will now define the basic operators using the following notation: x and y denote actions while P and Q denote processes. In addition, for each of the operators we include an example of an FSP expression using such operator as well as the graphical representation of the model described by that expression. For a full description of FSP syntax and semantics refer to [60].

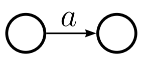
- **Primitive Process ``STOP''**: FSP has the primitive local process `STOP` which is a process that cannot engage in any action.



```
A = STOP.  A:
```

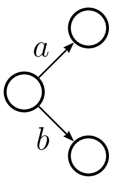
- **Action Prefix** ‘‘`->`’’: `( x -> P )` describes a process that initially engages in the action `x` and then behaves exactly as described by `P`.

```
A = ( a -> STOP ).  A:
```



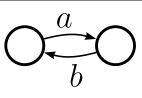
- **Choice** ‘‘`|`’’: `( x -> P | y -> Q )` describes a process which initially engages in either action `x` or `y`. If the first action is taken then the subsequent behaviour is described by `P` while if the second action is taken then `Q` describes the subsequent behaviour.

```
A = ( a -> STOP | b -> STOP ).   A:
```



- **Recursion**: the behaviour of a process can be defined recursively. The recursion may be directly in terms of the process being defined, or indirectly in terms of other processes.

```
A = ( a -> B ), B = ( b -> A).   A:
```



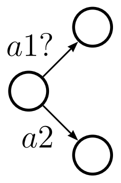
In order to be able to also define MTSs we have developed an extension to FSP. The idea is to allow the use of question marks in such a way that if an action's name includes at least one question mark then the corresponding transition should be interpreted as a maybe one. Otherwise, the action represents a required transition.

The labels of the MTS model are obtained by removing from the actions' names all occurrences of the question mark symbol. For example, `( topup?  -> STOP`

`)` represents a maybe transition through the topup label. It can also be written using the following notation: `( top?up -> STOP )`. The decision of allowing the inclusion of question marks in any part of the action's name apart from the first symbol is due to the fact that some advanced FSP operators generate actions with suffixes. Therefore, it is not possible to guarantee that if a question mark is included in the name it will be its last symbol.

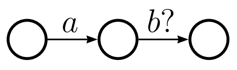
The following example represents a simple MTS described by an FSP expression.

```
A = ( a?1 -> STOP | a2 -> STOP ).   A:
```

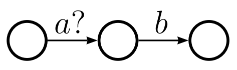


- **Parallel Composition** `‘‘||’’`: The Parallel Composition between models (Definition 6.1.1) is defined with the || operator. FSP syntax requires to prefix the name of the model with a "||" when the definition of the model uses previously defined models.

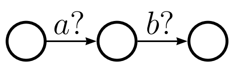
```
A = ( a -> b?  -> STOP).   A:
```



```
B = ( a?  -> b -> STOP).   B:
```



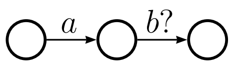
```
         ||PC = ( A || B).   PC:
```



- **Common Refinement** `‘‘`$+_{cr}$`’’`: The $+_{cr}$ operator (Definition 4.2.12) can be applied between models in order to define a new model. Following the FSP syntax in order to define a model that surges as an operation between previously defined models the prefix “||” has to be used in front of the name of the model.

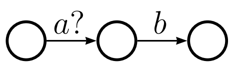
```
A = ( a -> b?  -> STOP).
```

A:



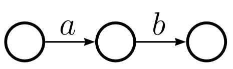
```
B = ( a?  -> b -> STOP).
```

B:



```
||CR = ( A +cr B).
```

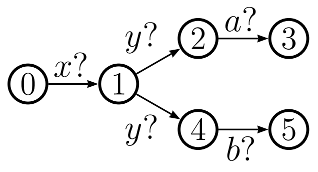
CR:



- **Merge** `‘‘++’’`: The Merge algorithm (Algorithm 4.2.17) can be applied using the ++ operator.

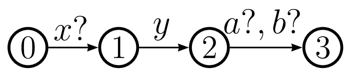
```
A = ( x?  -> A1),
A1 = ( y?  -> a?  -> STOP |
y?  -> b?  -> STOP).
```

A:



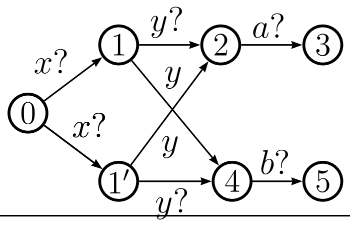
```
B = (x?  -> y -> { a?,b?  }  -> STOP).
```

B:



```
||M = ( A ++ B).
```

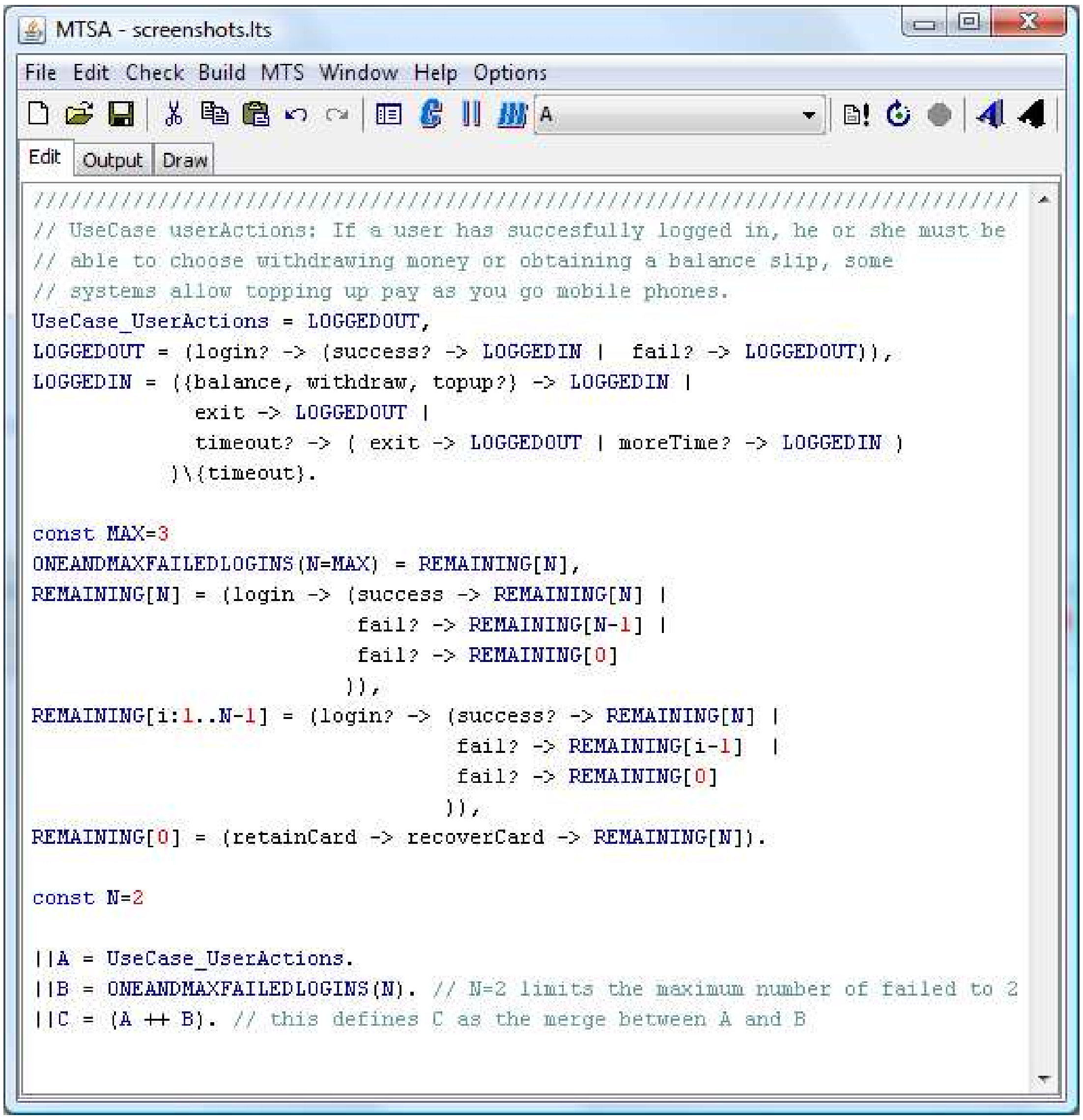
M:



7.2 Modal Transition System Analyser

In this section we show how to use MTSA and how to apply the main algorithms that we presented along this work. We divided the presentation in three main use cases: defining models and applying the merge operator, checking refinement, and checking consistency.

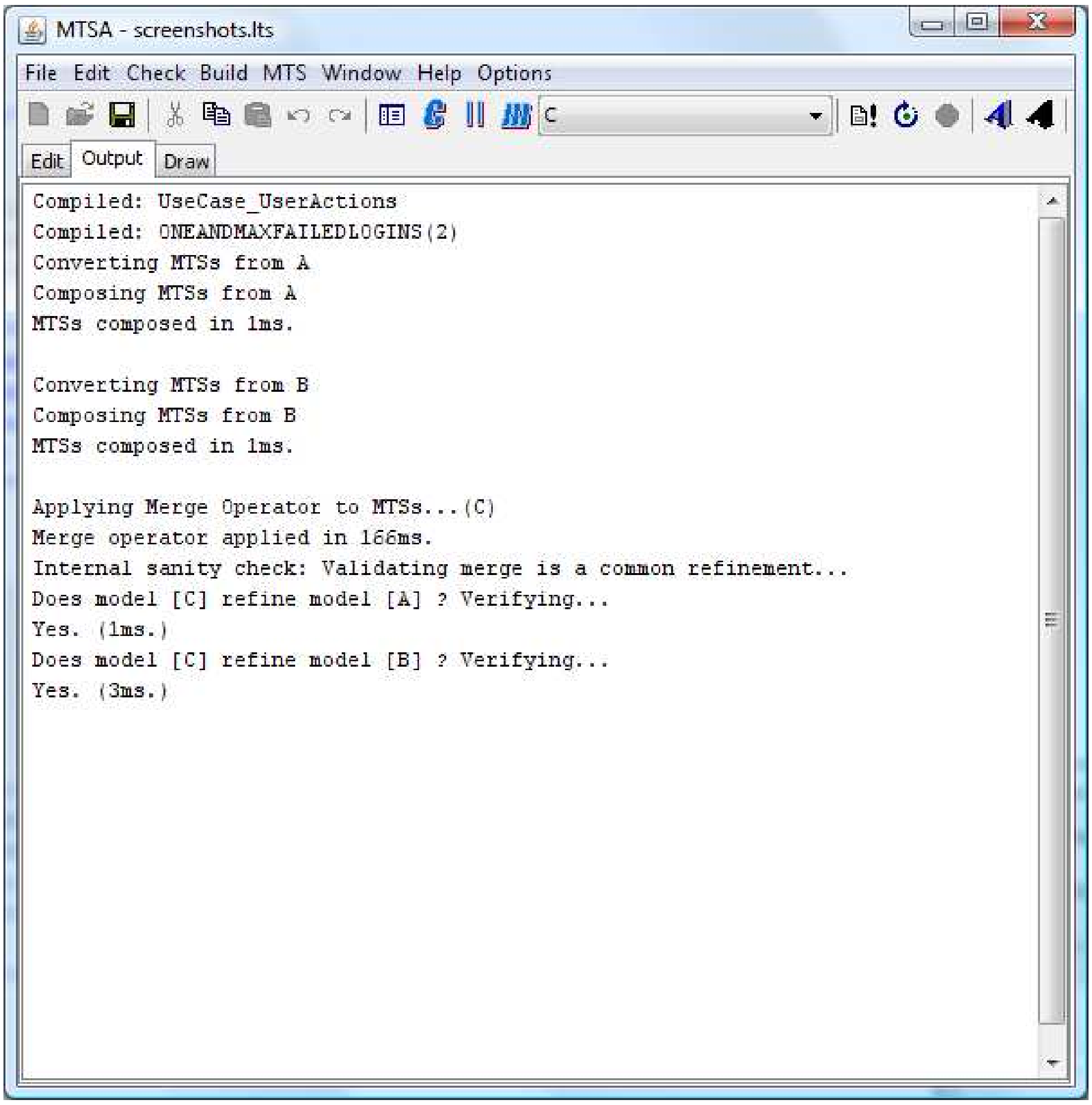
7.2.1 Model Definition and Merge



```
/////////////////////////////////////////////////////////////////////////////////
// UseCase userActions: If a user has succesfully logged in, he or she must be
// able to choose withdrawing money or obtaining a balance slip, some
// systems allow topping up pay as you go mobile phones.
UseCase_UserActions = LOGGEDOUT,
LOGGEDOUT = (login? -> (success? -> LOGGEDIN |  fail? -> LOGGEDOUT)),
LOGGEDIN = ({balance, withdraw, topup?} -> LOGGEDIN |
             exit -> LOGGEDOUT |
             timeout? -> ( exit -> LOGGEDOUT | moreTime? -> LOGGEDIN )
           )\{timeout}.

const MAX=3
ONEANDMAXFAILEDLOGINS(N=MAX) = REMAINING[N],
REMAINING[N] = (login -> (success -> REMAINING[N] |
                          fail? -> REMAINING[N-1] |
                          fail? -> REMAINING[0]
                         )),
REMAINING[i:1..N-1] = (login? -> (success? -> REMAINING[N] |
                                  fail? -> REMAINING[i-1]  |
                                  fail? -> REMAINING[0]
                                 )),
REMAINING[0] = (retainCard -> recoverCard -> REMAINING[N]).

const N=2

||A = UseCase_UserActions.
||B = ONEANDMAXFAILEDLOGINS(N). // N=2 limits the maximum number of failed to 2
||C = (A ++ B). // this defines C as the merge between A and B
```

Figure 7.2: MTSA *Edit* panel with the FSP definition for models $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ of the running example.

In Figure 7.2 we can see a screenshot of the MTSA tool. In this figure the *Edit* panel is selected. This panel is where the user can introduce model definitions using FSP as described in section 7.1. In this particular example, we can see the FSP definition of models $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ of our ATM running example (these models are depicted in Figure 1.1). Models $\mathcal{A}$ and $\mathcal{B}$ have been explicitly defined using FSP, while model $\mathcal{C}$ is the result of applying the merge algorithm (Algorithm 4.2.17) to models $\mathcal{A}$ and $\mathcal{B}$. After defining the models in the *Edit* panel the user can compile them to generate the models, and then be able to visualise and operate with the models. In Figure 7.3 we can see the *Output* panel that displays the result of the compilation process. In case that there are any errors during the compilation process those errors would be display in this panel.



```
Compiled: UseCase_UserActions
Compiled: ONEANDMAXFAILEDLOGINS(2)
Converting MTSs from A
Composing MTSs from A
MTSs composed in 1ms.

Converting MTSs from B
Composing MTSs from B
MTSs composed in 1ms.

Applying Merge Operator to MTSs...(C)
Merge operator applied in 166ms.
Internal sanity check: Validating merge is a common refinement...
Does model [C] refine model [A] ? Verifying...
Yes. (1ms.)
Does model [C] refine model [B] ? Verifying...
Yes. (3ms.)
```

Figure 7.3: MTSA *Output* panel after compiling the models.

In Figure 7.4 we can see the *Draw* panel, where the different models can be visualised.

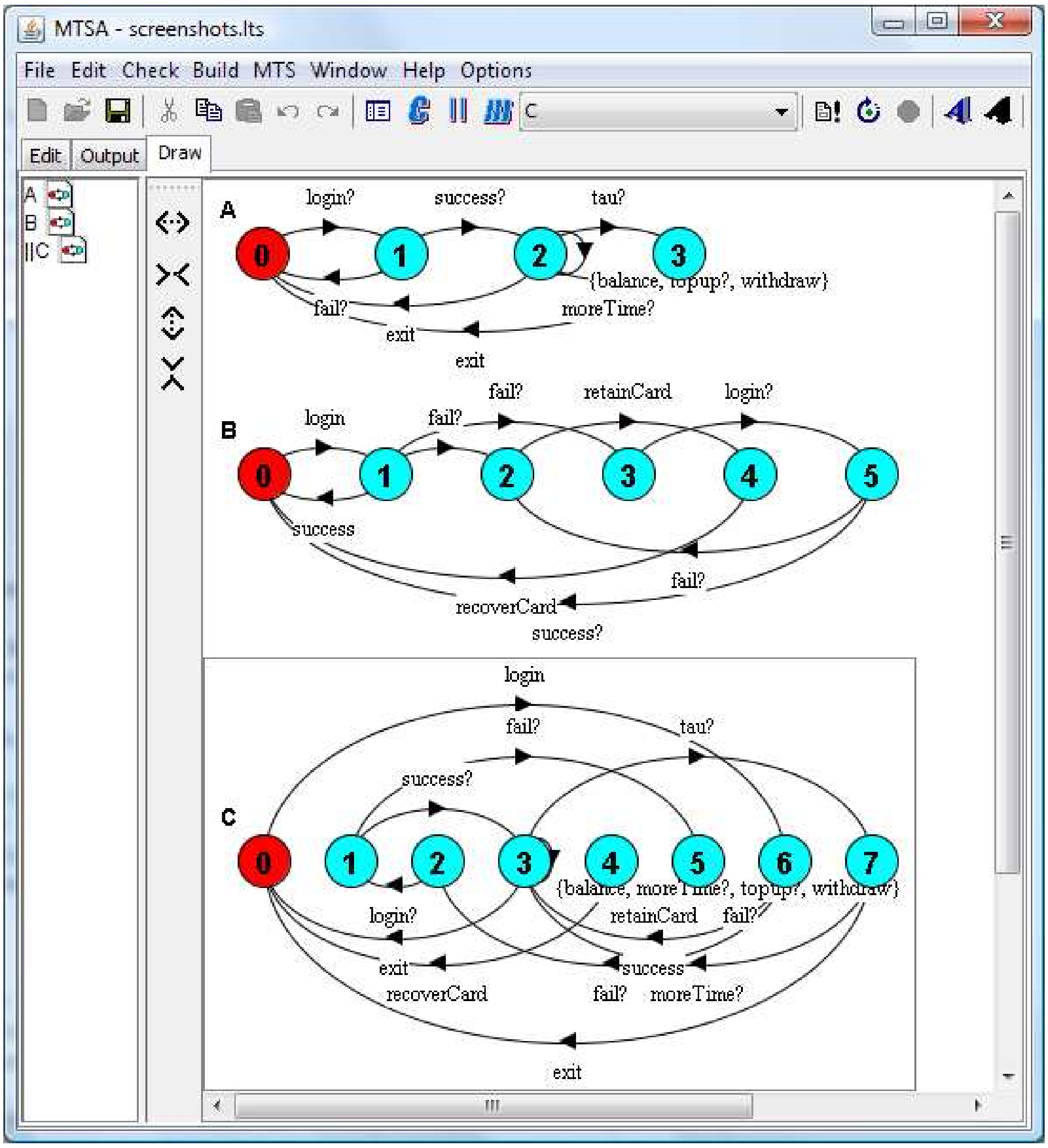


Figure 7.4: MTSA *Draw* panel displaying models $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$.

7.2.2 Refinement

In order to check refinement between two models the user should first define and compile those models. Once the models are generated the user can check refinement by selecting *Refinement* from the *MTS* menu. Figure 7.5 shows the dialog window that allows the user to select which models to check and the desired semantics. MTSA checks refinement using a fix-point algorithm to calculate the corresponding refinement relation. We have implemented a parallel version of the fix-point algorithm that leverages all the available cores in the system making a better use of the available resources and reducing the run time of the algorithm almost proportionally to the number of cores.

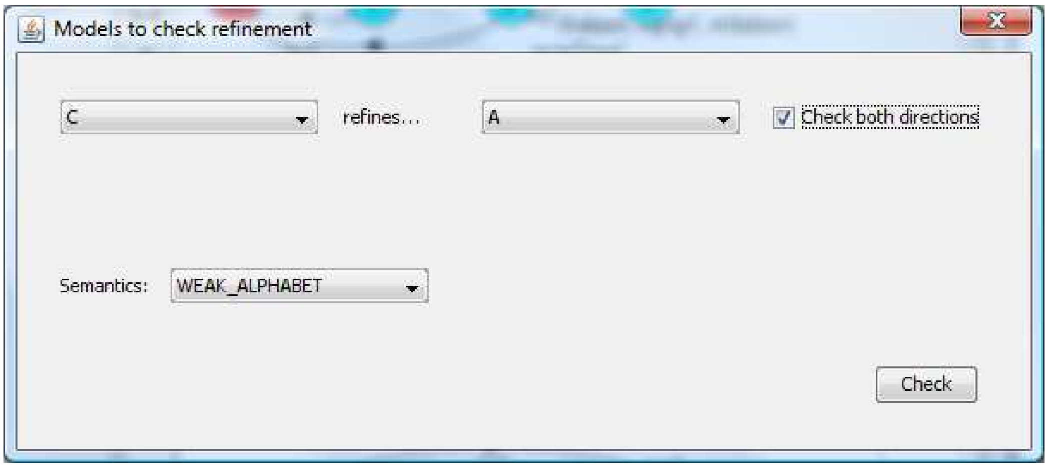
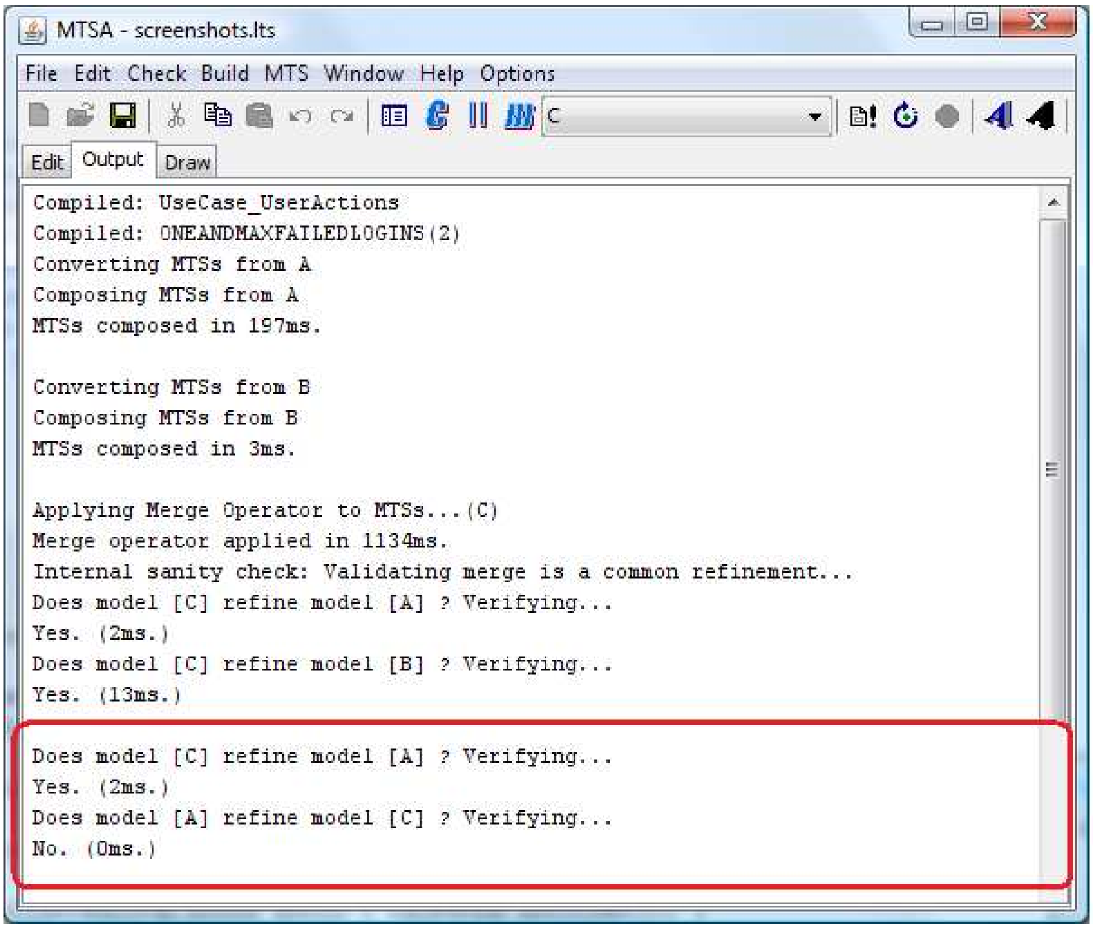


Figure 7.5: MTSA *Refinement* dialog window, with the options set to check weak alphabet refinement between models $\mathcal{A}$ and $\mathcal{C}$.

In Figure 7.6 we can see how MTSA displays the results of the refinement check in the *Output* panel.

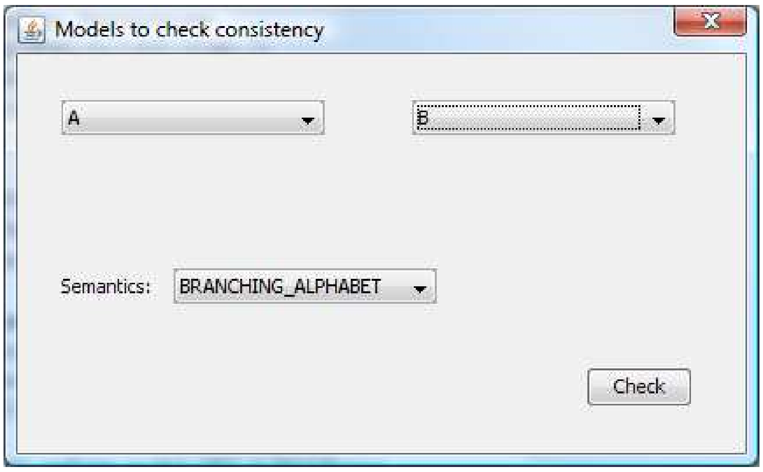


```
Compiled: UseCase_UserActions
Compiled: ONEANDMAXFAILEDLOGINS(2)
Converting MTSs from A
Composing MTSs from A
MTSs composed in 197ms.

Converting MTSs from B
Composing MTSs from B
MTSs composed in 3ms.

Applying Merge Operator to MTSs...(C)
Merge operator applied in 1134ms.
Internal sanity check: Validating merge is a common refinement...
Does model [C] refine model [A] ? Verifying...
Yes. (2ms.)
Does model [C] refine model [B] ? Verifying...
Yes. (13ms.)

Does model [C] refine model [A] ? Verifying...
Yes. (2ms.)
Does model [A] refine model [C] ? Verifying...
No. (0ms.)
```

Figure 7.6: Results of checking weak alphabet refinement between models $\mathcal{A}$ and $\mathcal{C}$, and vice versa.

7.2.3 Consistency

The steps to check consistency between two models are similar to those followed to check refinement. From the *MTS* menu the user selects *Consistency* and a dialog window that allows the user to select the models to be checked and the desired semantics appears (Figure 7.7).


Models to check consistency

A

B

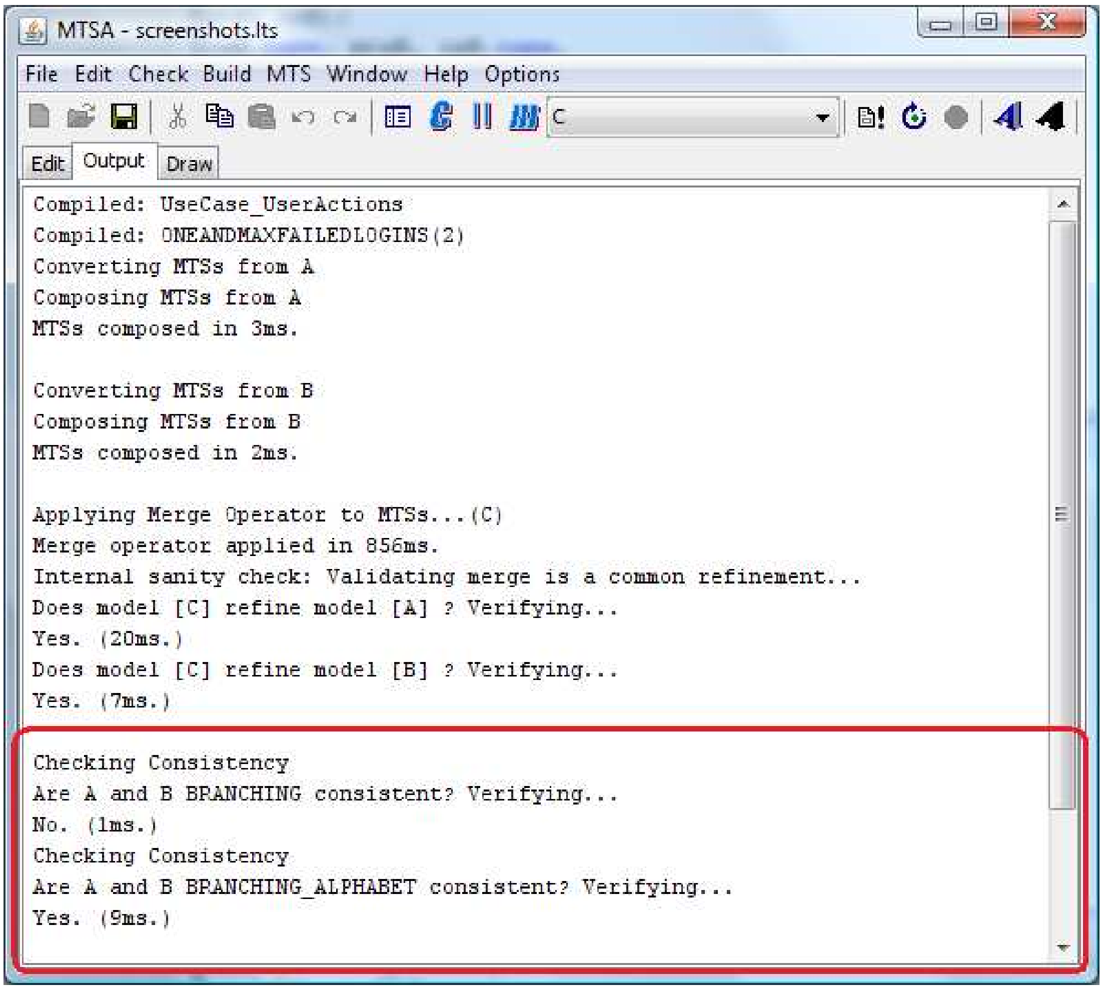
Semantics: BRANCHING_ALPHABET

Figure 7.7: MTSA *Consistency* dialog window, with the options set to check branching alphabet consistency between $\mathcal{A}$ and $\mathcal{B}$.

Internally, in order to check consistency the tool uses the same parallel fix-point algorithm that is used for checking refinement, but parameterised with the "simulation" rules for consistency. The fix-point algorithm was implemented in such a way that a user can easily implement different "simulation" relations between models and calculate them using this parallel fix-point algorithm.

In MTSA we have implemented the algorithms to calculate consistency relations for each of the semantics included in this thesis. It is worth noting that in the case of Weak Alphabet semantics, where the consistency relation we proposed is not complete, the algorithm we have implemented is the one presented in 4.2.11, which is more adequate for this semantics, as described in Section 4.2.2.

Finally, in Figure 7.8 we show the *Output* panel with the results of checking Branching and Branching Alphabet consistency between models $\mathcal{A}$ and $\mathcal{B}$.



```
Compiled: UseCase_UserActions
Compiled: ONEANDMAXFAILEDLOGINS(2)
Converting MTSs from A
Composing MTSs from A
MTSs composed in 3ms.

Converting MTSs from B
Composing MTSs from B
MTSs composed in 2ms.

Applying Merge Operator to MTSs...(C)
Merge operator applied in 856ms.
Internal sanity check: Validating merge is a common refinement...
Does model [C] refine model [A] ? Verifying...
Yes. (20ms.)
Does model [C] refine model [B] ? Verifying...
Yes. (7ms.)

Checking Consistency
Are A and B BRANCHING consistent? Verifying...
No. (1ms.)
Checking Consistency
Are A and B BRANCHING_ALPHABET consistent? Verifying...
Yes. (9ms.)
```

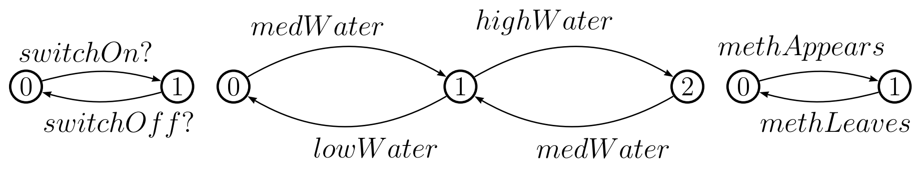
Figure 7.8: Results of checking branching alphabet consistency between $\mathcal{A}$ and $\mathcal{B}$.

Chapter 8

Case Study - Mine Pump

The purpose of this chapter is to show, by means of a case study, how the results described earlier in this thesis are applied in the context of an incremental behaviour model elaboration process. We base our analysis on the Mine Pump [59] case study, in which a pump controller is used to prevent the water in a mine pump from passing some threshold, and hence flooding the mine. To avoid explosions, the pump may only be active when there is no methane gas present in the mine. The pump controller monitors the water and methane levels by communicating with two sensors.

All analyses for this case study were performed automatically by means of the MTSA tool described in Chapter 7. It is worth noting that, while we only include as part of this chapter figures of the most relevant models (either in a graphical or textual representation), all models mentioned in this case study are defined in an FSP file that can be found at [29], and they can be visualised using the MTSA tool.



(a) Pump (b) WaterLevelSensor (c) MethaneSensor

Figure 8.1: The LTSs for (a) *Pump*, (b) *WaterLevelSensor*, and (c) *MethaneSensor*.

This chapter is organised as follows. In Section 8.1, we provide a high level description of the Mine Pump System and its components. In Section 8.2, we focus on the most complex component of the system, the Mine Pump Controller, and show how it can be constructed by merging several partial models and how tool-supported validation of the resulting model can prompt further elaboration. We then construct the final model of the Mine Pump System, which satisfies the expected requirements, through successive merge operations over partial models. In Section 8.3 we show how the Mine Pump System can be extended as new requirements are identified. Finally, in Section 8.4 we discuss the results.

8.1 Mine Pump System Description

The mine pump system consists of four components: *Pump*, *PumpController*, *WaterLevelSensor*, and *MethaneSensor*. The complete system, *MinePumpSystem*, is the parallel composition of these components. The component *Pump* models the physical pump, which can be switched on and off. *PumpController* describes the controller that monitors the water and methane levels, and controls the pump in order to guarantee the correct behaviour for the mine pump system. *WaterLevelSensor* models the water sensor and includes assumptions on how the water level is expected to change between low, medium, and high. *MethaneSensor* keeps track of whether methane is present in the mine.

The LTSs for the *Pump*, *WaterLevelSensor*, and *MethaneSensor* components are shown in Figure 8.1, where we assume that initially the water is low, the pump is off, and no methane is present.

The description given above leaves open the exact water level at which to turn the pump on and off. For example, the pump could be turned on when there is high water or possibly when the water is not low, (e.g., at a medium level). The pump could be turned off when there is low water or possibly when the water is not high. In what follows, we investigate in more detail partial models for the pump controller, which are intended to be merged to create a model of the entire controller, namely, *PumpController*.

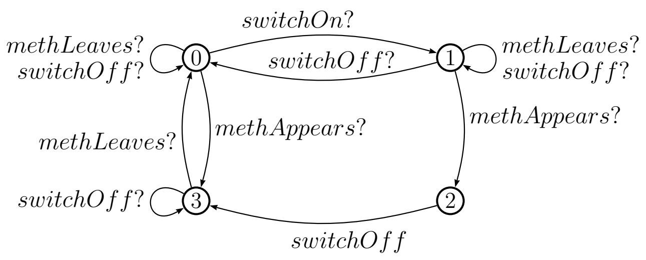
8.2 Pump Controller Construction and Elaboration

8.2.1 Initial Partial Models

Assume that requirements specification of the pump controller has been organized following the IEEE Recommended Practice for Software Requirements Specifications Standard 830 [53], which provides a template for structuring requirements based on the operation mode of the system-to-be. Consequently, requirements are grouped for example into those that are relevant when the mine pump is on and those in which the mine pump is off.

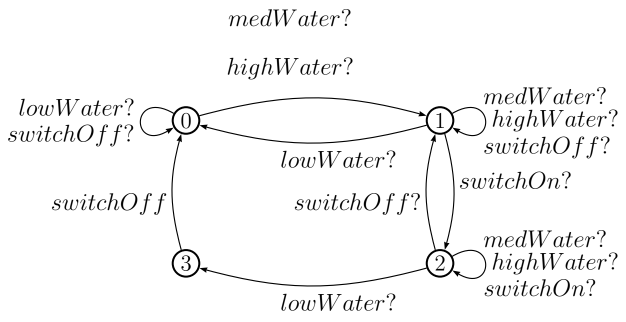
As with the ATM running example, the operational requirements for the mine pump controller could be given in a variety of specification languages. From some of these languages (e.g., MSCs, use cases or temporal logics), MTS models could be synthesized automatically [88, 89]. Synthesis of MTSs is beyond the scope of this thesis and consequently of this case study. Hence, we assume that the MTSs have been constructed manually or (semi-)automatically from the requirements corresponding to each mode.

We model the mine pump controller giving three descriptions of the controller from different points of view, that capture the initial requirements that have been gathered. Each of these descriptions is a partial description and focuses on one aspect of the controller behaviour, and might have been provided by different stakeholders. These descriptions are modelled with MTSs and each model characterises the set of all controllers which manifest the described behaviour. The three models are: *OnPolicy*, *OffPolicy*, and *SafetyPolicy*. The complete controller, *PumpController*, is the merge of these models.



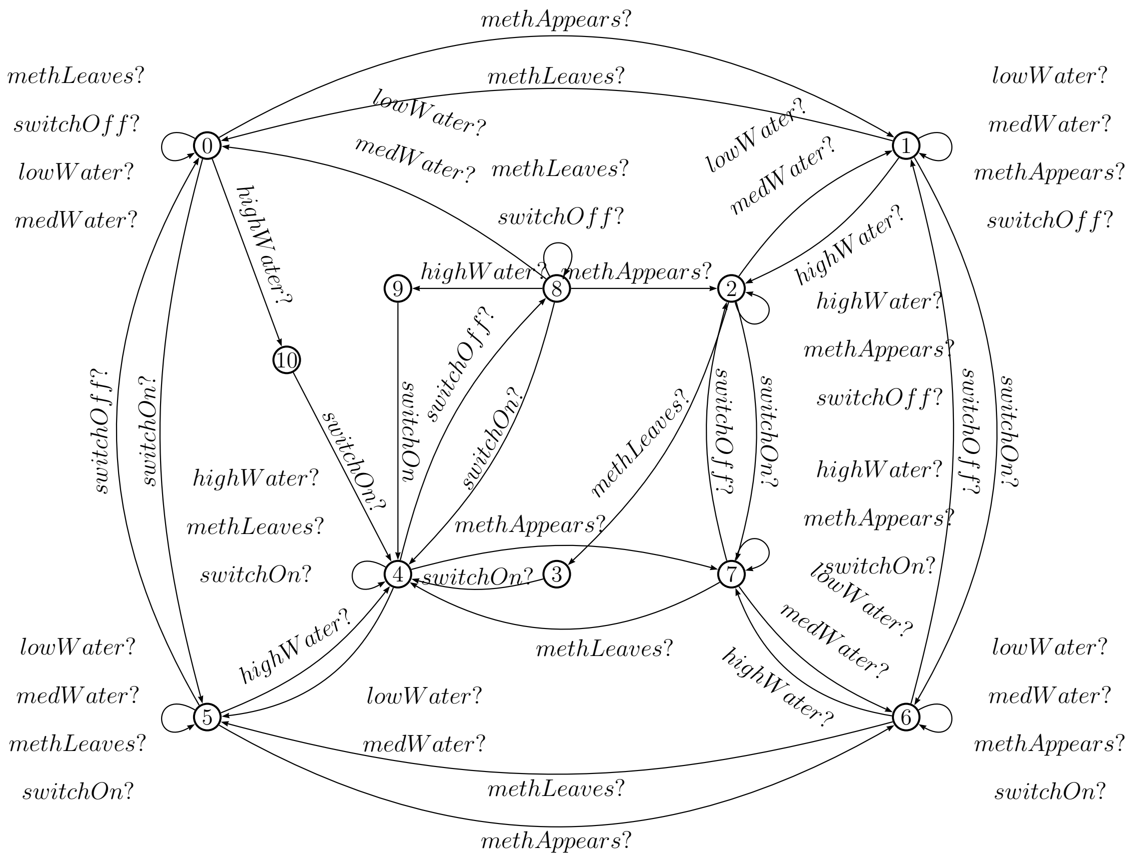
```
SafetyPolicy = PUMP[False][False],
  PUMP[methane:Bool][pumpOn:Bool] = (
    switchOff? -> PUMP[methane][False] |
    methLeaves? -> PUMP[False][pumpOn] |
    when (!methane && pumpOn)  methAppears? -> switchOff -> PUMP[True][False] |
    when (!methane && !pumpOn) methAppears? -> PUMP[True][False] |
    when (!methane) switchOn? -> PUMP[False][True]
  ).
```

Figure 8.2: FSP and graphical representation of the MTS of the Safety Policy.



```
OffPolicy = PUMP[True][False],
  PUMP[lowWater:Bool][pumpOn:Bool] = (
    when (!lowWater) switchOn?  -> PUMP[lowWater][True] |
    switchOff? -> PUMP[lowWater][False] |
    {medWater?,highWater?} -> PUMP[False][pumpOn] |
    when (pumpOn) lowWater? -> ( switchOff -> PUMP[True][False] ) |
    when (!pumpOn) lowWater? -> PUMP[True][False]
  ).
```

Figure 8.3: FSP and graphical representation of the MTS of the Off Policy.



```
OnPolicy = PUMP[False][False][False],
  PUMP[highWater:Bool][methane:Bool][pumpOn:Bool] = (
    methAppears? -> PUMP[highWater][True][pumpOn] |
    switchOff? -> PUMP[highWater][methane][False] |
    switchOn? -> PUMP[highWater][methane][True] |
    {lowWater?,medWater?} -> PUMP[False][methane][pumpOn] |
    when (methane)
        highWater? -> PUMP[True][True][pumpOn] |
    when (!methane && !pumpOn)
        highWater? -> ( switchOn -> PUMP[True][False][True]) |
    when (!methane &&  pumpOn)
        highWater? -> PUMP[True][False][True] |
    when (methane && highWater && !pumpOn)
        methLeaves? -> ( switchOn -> PUMP[True][False][True] ) |
    when (!methane || !highWater || pumpOn)
        methLeaves? -> PUMP[highWater][False][pumpOn]
  ).
```

Figure 8.4: FSP and graphical representation of the MTS of the On Policy.

OnPolicy describes when the pump must be turned on to avoid flooding the mine, leaving open the option of turning the pump on with a lower level of water. It specifies that the pump must be turned on if the water level is high and there is no methane. The pump might be damaged if it works with not enough water to pump out. Therefore the controller has to protect the pump and guarantee that the pump will not work with a low level of water. This property is captured with *OffPolicy* and leaves open the option of turning the pump off with a higher level of water. *SafetyPolicy* captures the requirement that the pump must be kept off under the presence of methane in order to prevent an explosion in the mine. This model only specifies that the pump must be turned off immediately when methane appears and that the pump cannot be turned on if there is methane present in the mine, leaving open when the controller should turn the pump on or off when there is no methane present.

The MTS obtained from the merge of the described models characterises the set of LTSs that fulfil the mentioned properties. As mentioned before, the requirements captured so far do not specify the exact water level at which to turn the pump on and off, leaving these decisions open for a further refinement based on gathering more detailed requirements or an arbitrary election of one implementation among all the possible implementations.

In Figures 8.2, 8.3 and 8.4 we can see the described models. It is important to highlight that all these models are a partial description of the whole controller, and they have been modelled using only the alphabet which is relevant for the behaviour that each of them is describing. Being able to use only the alphabet which is relevant to each model allows us to produce simpler and more compact models. This is possible because we are using branching alphabet semantics. In [33] we presented a predecessor to this case study. At that time, we were limited to applying the merge algorithm under strong semantics only and therefore the whole alphabet had to be embedded in all models leading to more complex and harder to follow models.

Our goal is to build an MTS which characterises all the LTSs which satisfy all the requirements that we have captured for the controller so far. This could be achieved by combining the three models that we have created.

8.2.2 Merge vs. $+_{cr}$ operator

Now that we have a set of MTSs that describe different aspects of the pump controller we would like to combine them in order to get a more precise and consolidated description of the behaviour of the controller. As we have been arguing along this work the merge operation provides the right support to do so, while previous approaches might be insufficient since they fail to obtain the LCR or a common refinement which is abstract enough to carry on with the elaboration process.

If we attempt to combine the *OnPolicy*, *OffPolicy* and *SafetyPolicy* using the $+_{cr}$ operator instead of the merge, we would not be able to get a common refinement of our three models for the pump controller. For example, if we apply the $+_{cr}$ first to *OnPolicy* and *OffPolicy*, the result that we get is not the LCR between these models but a more refined model. The resulting model is inconsistent with *SafetyPolicy*, which prevents the elaboration process to continue. If we try the other two possible ways of combining these models with the $+_{cr}$ operator we will

```
PumpController = Q0,
Q0  = ({lowWater?, methLeaves?,
    switchOff?} -> Q0
    |methAppears? -> Q1
    |medWater? -> Q2
    |highWater? -> Q3),
Q1  = (methLeaves? -> Q0
    |{lowWater?, switchOff?} -> Q1
    |highWater? -> Q9
    |medWater? -> Q10),
Q2  = (lowWater? -> Q0
    |{medWater?, methLeaves?,
        switchOff?} -> Q2
    |highWater? -> Q3
    |switchOn? -> Q6
    |methAppears? -> Q10),
Q3  = (switchOn -> Q4),
Q4  = ({highWater?, methLeaves?,
    switchOn?} -> Q4
    |lowWater? -> Q5
    |medWater? -> Q6
    |switchOff? -> Q7
    |methAppears? -> Q8),
Q5  = (switchOff -> Q0
    |methLeaves? -> Q5),
Q6  = (switchOff? -> Q2
    |highWater? -> Q4
    |lowWater? -> Q5
    |{medWater?, methLeaves?,
        switchOn?} -> Q6
    |methAppears? -> Q11),
Q7  = (lowWater? -> Q0
    |medWater? -> Q2
    |highWater? -> Q3
    |switchOn? -> Q4
    |{methLeaves?, switchOff?} -> Q7
    |methAppears? -> Q9),
Q8  = (highWater? -> Q8
    |switchOff -> Q9
    |medWater? -> Q11),
Q9  = (lowWater? -> Q1
    |methLeaves? -> Q3
    |{highWater?, switchOff?} -> Q9
    |medWater? -> Q10),
Q10 = (lowWater? -> Q1
    |methLeaves? -> Q2
    |highWater? -> Q9
    |{medWater?, switchOff?} -> Q10),
Q11 = (highWater -> Q8
    |switchOff? -> Q10
    |medWater? -> Q11).
```

Figure 8.5: First model for the *PumpController* resulting from merging *SafetyPolicy*, *OnPolicy*, and *OffPolicy*.

find that all of them fail to produce a common refinement of the three models.

If we now try to combine our three models for the pump controller using the merge operator we do get a branching alphabet common refinement regardless of the order in which we apply the merge operator. It is worth noting that in this case, the order in which the models are merged together leads to different MCRs. This is in line with Property 6.3.2 and the discussion we presented in Section 6.3 in relation to this property. In Figure 8.5 we can see the result of (*SafetyPolicy*++*OnPolicy*)++*OffPolicy*. This operation returns an MTS with maybe transitions, implying that there are many LTSs which satisfy all the requirements. This indicates that we can continue the elaboration process of the pump controller by eliciting more requirements, or alternatively we can choose at this stage one of the valid implementations of the model that we have built so far.

```
ImplOff = Q0,
  Q0  = ({lowWater, medWater,
      methLeaves, switchOff} -> Q0
      |highWater -> Q1
      |methAppears -> Q5),
  Q1  = (switchOn -> Q2),
  Q2  = (switchOff -> Q3),
  Q3  = ({lowWater, medWater} -> Q0
      |{methLeaves, switchOff} -> Q3
      |methAppears -> Q4),
  Q4  = (methLeaves -> Q1
      |{highWater, switchOff} -> Q4
      |{lowWater, medWater} -> Q5),
  Q5  = (methLeaves -> Q0
      |highWater -> Q4
      |{lowWater, medWater, switchOff}
         -> Q5).
```

Figure 8.6: Valid implementation for the pump controller depicted in Figure 8.5.

8.2.3 Further Elaboration of the Pump Controller

We will continue the elaboration process by eliciting new requirements. We can drive this process by analysing possible implementations of the current model with the stakeholders of the system. Figure 8.6 shows a valid implementation of the model for the *PumpController* presented in Figure 8.5, hence it is also a valid implementation of *SafetyPolicy*, *OnPolicy*, and *OffPolicy*, as we can check with the MTSA tool. This implementation fulfils the requirements that we have captured so far for the pump controller. However, if we look at state 2 we can see that this potential controller always turns the pump off immediately after it turns the pump on. Analysing the behaviour of this implementation allows us to extract the following new requirement:

- [*OffPolicy2*] The pump should not be turned off with a high water level except if methane appears.

By analysing the MTS for the pump controller and the different maybe transitions it presents, the following additional requirements were also identified:

- [*OnPolicy2*] The pump should only be turned on if the water level is high.
- [*OffPolicy3*] The pump should be turned off as soon as the water level is below high.

```
PumpController = Q0,
Q0  = ({lowWater?, methLeaves?,
    switchOff?} -> Q0
    |methAppears? -> Q1
    |medWater? -> Q2
    |highWater? -> Q3),
Q1  = (methLeaves? -> Q0
    |{lowWater?, switchOff?} -> Q1
    |highWater? -> Q9
    |medWater? -> Q10),
Q2  = (lowWater? -> Q0
    |{medWater?, methLeaves?,
        switchOff?} -> Q2
    |highWater? -> Q3
    |switchOn? -> Q6
    |methAppears? -> Q10),
Q3  = (switchOn -> Q4),
Q4  = ({highWater?, methLeaves?,
    switchOn?} -> Q4
    |lowWater? -> Q5
    |medWater? -> Q6
    |switchOff? -> Q7
    |methAppears? -> Q8),
Q5  = (switchOff -> Q0
    |methLeaves? -> Q5),
Q6  = (switchOff? -> Q2
    |highWater? -> Q4
    |lowWater? -> Q5
    |{medWater?, methLeaves?,
        switchOn?} -> Q6
    |methAppears? -> Q11),
Q7  = (lowWater? -> Q0
    |medWater? -> Q2
    |highWater? -> Q3
    |switchOn? -> Q4
    |{methLeaves?, switchOff?} -> Q7
    |methAppears? -> Q9),
Q8  = (highWater? -> Q8
    |switchOff -> Q9
    |medWater? -> Q11),
Q9  = (lowWater? -> Q1
    |methLeaves? -> Q3
    |{highWater?, switchOff?} -> Q9
    |medWater? -> Q10),
Q10 = (lowWater? -> Q1
    |methLeaves? -> Q2
    |highWater? -> Q9
    |{medWater?, switchOff?} -> Q10),
Q11 = (highWater -> Q8
    |switchOff? -> Q10
    |medWater? -> Q11).
```

Figure 8.7: Second model for the *PumpController* resulting from merging *SafetyPolicy*, *OnPolicy*, *OffPolicy*, *OnPolicy2*, *OffPolicy2*, and *OffPolicy3*.

To build a new model for the controller which satisfies all the requirements that have been captured so far, we refine the *PumpController* model, merging it with the models which capture the new requirements. Figure 8.7 presents the MTS that results from this operation. As we can see, it is an MTS with maybe transitions. If we compose in parallel the new model for the pump controller with the models of the environment (*Pump*, *WaterLevelSensor*, and *MethaneSensor*) we obtain an MTS for the whole system. This model of the entire system still has maybe transitions. However, all the maybe transitions are on actions from the environment (*lowWater*, *medWater*, *highWater*, *methAppears*, and *methLeaves*) and there are no maybe transitions on any of the controllable actions (*switchOn*, *switchOff*). This indicates that in fact there are no more possible refinements to do on the pump controller that would change how the pump is controlled. The only possible refinement that we can do on the pump controller would be adding

```
PumpControllerImpl = Q0,
  Q0  = ({lowWater, medWater,
      methLeaves, switchOff} -> Q0
      |highWater -> Q1
      |methAppears -> Q6),
  Q1  = (switchOn -> Q2),
  Q2  = ({highWater, methLeaves,
      switchOn} -> Q2
      |{lowWater, medWater} -> Q3
      |methAppears -> Q4),
  Q3  = (switchOff -> Q0
      |methLeaves -> Q3),
  Q4  = (highWater -> Q4
      |switchOff -> Q5),
  Q5  = (methLeaves -> Q1
      |{highWater, switchOff} -> Q5
      |{lowWater, medWater} -> Q6),
  Q6  = (methLeaves -> Q0
      |highWater -> Q5
      |{lowWater, medWater, switchOff}
         -> Q6).
```

Figure 8.8: Implementation for the *PumpController* obtained as a result of the elaboration process.

```
MinePumpSystem = Q0,
  Q0  = (medWater -> Q1
      |methAppears -> Q8),
  Q1  = (lowWater -> Q0
      |highWater -> Q2
      |methAppears -> Q7),
  Q2  = (switchOn -> Q3),
  Q3  = (medWater -> Q4
      |methAppears -> Q5),
  Q4  = (switchOff -> Q1),
  Q5  = (switchOff -> Q6),
  Q6  = (methLeaves -> Q2
      |medWater -> Q7),
  Q7  = (methLeaves -> Q1
      |highWater -> Q6
      |lowWater -> Q8),
  Q8  = (methLeaves -> Q0
      |medWater -> Q7).
```

Figure 8.9: Model for the *MinePumpSystem* that results from the parallel composition of *PumpControllerImpl*, *WaterLevelSensor*, *MethaneSensor*, and *Pump*.

assumptions or preconditions on how the environment behaves. Therefore, we will stop the elaboration process at this point taking the implementation that has the maximum behaviour possible, i.e. all maybe transitions are converted to required. In this way we are producing a controller that does not make any assumptions on the behaviour of the environment nor does it impose any particular preconditions on it. Figure 8.8 shows the model *PumpControllerImpl*, which is the implementation for the pump controller that we obtained with this elaboration process. Figure 8.9 shows the final model for the whole mine pump system that results from the parallel composition of *PumpControllerImpl*, *WaterLevelSensor*, *MethaneSensor*, and *Pump*.

```
AlarmPolicy = Q0,
  Q0  = ({lowWater?, medWater?,
      methLeaves?, switchOff?, switchOn
      ?} -> Q0
      |highWater? -> Q1
      |methAppears? -> Q8),
  Q1  = ({lowWater?, medWater?} -> Q0
      |{highWater?, methLeaves?,
          switchOff?, switchOn?} -> Q1
      |methAppears? -> Q2),
  Q2  = (highWater? -> Q2
      |switchOff -> Q3),
  Q3  = (alarmOn -> Q4),
  Q4  = ({highWater?, switchOff?,
      switchOn?} -> Q4
      |methLeaves? -> Q5
      |medWater? -> Q7
      |lowWater? -> Q10),
  Q5  = (switchOn -> Q6),
  Q6  = (alarmOff -> Q1),
  Q7  = (alarmOff -> Q8),
  Q8  = (methLeaves? -> Q0
      |{lowWater?, medWater?,
          methAppears?, switchOff?,
          switchOn?} -> Q8
      |highWater? -> Q9),
  Q9  = (alarmOn -> Q4),
  Q10 = (alarmOff -> Q8).
```

Figure 8.10: Model for the *AlarmPolicy*.

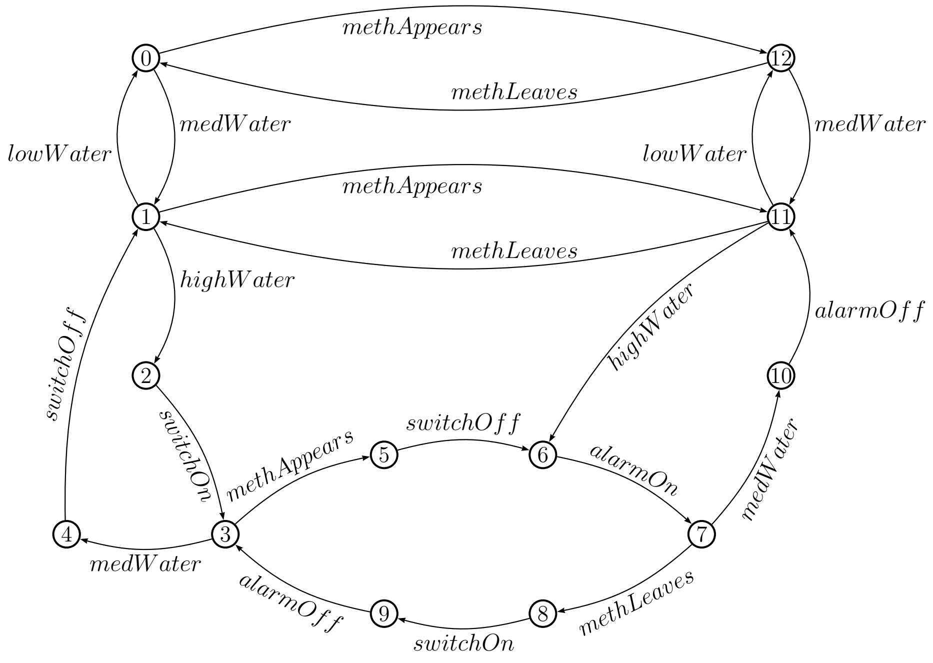
8.3 Extending the Mine Pump System

After the model *PumpControllerImpl* was built the stakeholders identified that the system could be in a state where the water level is high and there is methane in the mine, leading to a highly risky situation. Therefore, they requested to add to the system an *alarm* that should be on while the system is in the described condition, and off otherwise. Also, considering the risk of explosion in the mine if the pump is on and the presence of methane is detected, the pump should be turned off before activating the alarm if necessary.

To incorporate these new requirements we will add a new component *Alarm* to the system, which represents the physical alarm, and update the controller in order to handle not only the pump but also the alarm. The alarm is modelled similarly to the pump, but in this case we have the actions *alarmOn* and *alarmOff*. We will then compose the *Alarm* model with the rest of the models (*Pump*, *PumpController*, *WaterLevelSensor*, and *MethaneSensor*) to obtain a model for the whole system.

The controller needs to be extended to handle the alarm as required, while its behaviour keeps fulfilling the previous requirements. The need to control the alarm raises the need to expand the alphabet of the controller model with the labels *alarmOn* and *alarmOff*. In order to incorporate this new requirement into the controller we can follow the same elaboration process used with the initial requirements. We produce an MTS that captures the behaviour required to control the alarm while leaving open any other behaviour, and we then merge this new model with our existing model for the controller. In Figure 8.10 we can see the *AlarmPolicy* model that captures the requirements to control the alarm. As we can see, the model uses the new labels *alarmOn* and *alarmOff*, so merging this model with the existing pump controller, which does not have these labels, would not be supported under traditional semantics for MTSs. However, we can merge these models using branching alphabet semantics, producing a model which is not only a common refinement but also an alphabet extension.

Consider that an implementation for the controller had already been built based on the *PumpControllerImpl* model that we generated in the previous elaboration phase. In order to save implementation efforts it would be useful to question whether it is possible to extend the existing implementation with the *AlarmPolicy*. To answer this we need to check if the *PumpControllerImpl* is consistent with the *AlarmPolicy*. Using the MTSA tool we verify that these models are consistent, and therefore the implementation can be extended. In order to do so we merge it with the *AlarmPolicy* model. Once this step is completed, we compose in parallel the models of the environment with the extended implementation for the pump controller in order to generate the model for the complete system. Figure 8.11 shows the final mine pump system that we obtain after following this process.



```
SystemWithAlarm = Q0,                          Q6  = (alarmOn -> Q7),
  Q0  = (medWater -> Q1                        Q7  = (methLeaves -> Q8
      |methAppears -> Q12),                        |medWater -> Q10),
  Q1  = (lowWater -> Q0                        Q8  = (switchOn -> Q9),
      |highWater -> Q2                         Q9  = (alarmOff -> Q3),
      |methAppears -> Q11),                    Q10 = (alarmOff -> Q11),
  Q2  = (switchOn -> Q3),                      Q11 = (methLeaves -> Q1
  Q3  = (medWater -> Q4                            |highWater -> Q6
      |methAppears -> Q5),                         |lowWater -> Q12),
  Q4  = (switchOff -> Q1),                     Q12 = (methLeaves -> Q0
  Q5  = (switchOff -> Q6),                         |medWater -> Q11).
```

Figure 8.11: The final model for the entire *MinePumpSystem* with the alarm.

8.4 Discussion

In this case study we have shown how branching alphabet semantics allows us to support an iterative modelling process in a context where initially only partial information is available and new information becomes available as the project evolves. While traditional semantics would not be suited for incremental elaboration, branching alphabet semantics allows us to do this by extending the alphabet as we need to do so. This approach also eases the modelling process by allowing the modeller to construct many simpler partial models that capture individual requirements, which are easier to build as well as easier to validate. Furthermore, this case study shows how the consistency and merge algorithms presented in this thesis can be used to adequately support the elaboration process, improving on previously defined algorithms.

Chapter 9

Conclusions

In this chapter we first present a review of relevant related work, followed by an evaluation of the contributions presented in this thesis and suggested directions for future work.

9.1 Related Work

Below, we survey related work along three directions: (1) behaviour modelling, (2) consistency and merge, and (3) abstraction and property preservation with respect to partial models.

Behaviour Modelling

A significant body of work has been produced in the area of behaviour modelling, including research on process algebras (e.g., [45]), notions of equivalence and refinement (e.g., [70]), and model checking (e.g., [16]). The bulk of this work has used a two-valued semantics approach to behaviour modelling (e.g., using LTSs [58] as the underlying formalism). Typically, the behaviour explicitly described by the underlying state-machine is considered to be required, while the rest is considered to be prohibited. As stated previously, the assumption that the underlying state machine is complete, up to some level of abstraction, is limiting in the context of iterative development processes [9], and in processes that adopt use-case and scenario-based specifications (e.g., [19, 87]), or that are viewpoint-oriented [47].

While LTSs and other two-valued state machine formalisms can capture some notion of partiality, the behaviour they describe is considered as either the upper or the lower bound to the final, complete, system behaviour (see our discussion in Section 2.1), *but not both.* Partial behavioural formalisms capture this nicely, by capturing the unknown behaviour explicitly, so as new information becomes available, the two bounds can be refined simultaneously. In MTSs, this unknown behaviour is specified by transitions which are possible but not required.

A number of formalisms exist which allow explicit modelling of lack of information. Partial Kripke structures [11] and Kripke Modal Transition Systems [49] extend Kripke structures to support propositions in states to be one of three values (true, false, and *unknown*). In our work, states in themselves do not have any semantics, we focus only on observable system behaviour as described by the labelled transitions between states, hence we build on models in the labelled transition systems [58] style.

Our definition of Modal Transition Systems is essentially that proposed by Larsen et al. [66]. However, in [66] all MTSs have the same alphabet, the universe of all labels, while we extend the definition of MTSs to include a communication alphabet in line with [60]. Having the communication alphabet allows scoping models and capturing the fact that system components may control and monitor different sets of events [55].

In this work, we have focused on Modal Transition Systems which are less expressive than other partial behaviour modelling formalisms that have been proposed, such as multi-valued Kripke structures [14] and Mixed Transition Systems [22]. There is a trade-off between expressiveness, tractability and understandability and further studies, extending the results presented in this thesis to these formalisms, are necessary.

Numerous extensions of MTS exist such as Mixed Transition Systems [22] and Disjunctive Modal Transition Systems [63]. Antonik et. al. [2] present a survey which provides an excellent coverage of the relation between these different formalisms and other extensions. The novel semantics we proposed could be studied for these formalisms too. We believe that existing weak and strong refinement notions in these settings will suffer from the same shortcomings as MTSs. A slightly different approach to modelling unknown behaviour is taken in [85, 71]. In [85] the authors have studied Partial Labelled Transition Systems, where each state is associated with a set of actions that are explicitly proscribed from happening. Extended Transition Systems [71] also associate a set of actions with each state, but in this case it models the actions for which the state has been fully described. These models are special MTSs [51], the new notions of refinement introduced in this work and the merge operation have yet to be studied when restricted to these models.

A recent extension to MTSs has been presented by Bauer et al. [6], where MTSs are extended with structured labels that represent quantitative aspects of the models. A new refinement notion for these models is also presented, which extends the classical notion of strong refinement for MTS to include the capability to refine labels based on their pre-order in the label set. This notion of refinement is orthogonal to the alphabet refinement we presented in this work. While the label-structured refinement allows the modeller to gradually refine the label of one transition, the alphabet refinement we presented allows him to introduce new concepts into the model.

Consistency and Merge

Composition of behaviour models is not a novel idea [70, 45]; however, its main focus has been on *parallel* composition, which describes how two *different* components work together. In the context of model elaboration, we are interested in *merge*, i.e., composing two partial descriptions of the *same* component to obtain a model that is more comprehensive than either of the original partial descriptions.

The notion of merge in itself is not novel either; it underlies many approaches to system model elaboration such as viewpoints [20], aspects [15], and scenario/use case composition (e.g., [86, 61]). However, the interplay of partial descriptions and merge is not necessarily treated explicitly and formally.

Larsen et. al. originally introduced a merge operator (called *conjunction*), but defined it only for MTSs over the same vocabulary without τ transitions, and for which there is an *independence relation* (at which point the least common refinement exists) [65]. Their goal is to decompose a complete specification into several partial ones to enable compositional proofs. Although not studied in depth, the operator in [65] is based on strong refinement. In particular, [65, 62] use an incomplete notion of consistency and do not address the problem of multiple MCRs.

In [90] an initial study of merge and consistency for weak semantics is presented; however, the results presented in here are stronger. The subtleties of the existence of multiple MCRs under weak semantics were also initially discussed in [90] and then resolved as part of this work.

In [63] a conjunction operator for Disjunctive MTSs (DMTSs), similar to the one in [65], is defined. These models simplify merging by allowing inconsistencies of models being merged to be encoded within the DMTSs. However, the computational complexity of merging MTS is traded for the complexity of detecting contradictions: Checking that a DMTS has an implementation by inspection is non-trivial even in small examples and in general it is computationally as expensive as merge is in MTS. Checking consistency of an MTS is trivial as by definition any MTS has an implementation. The goal of [63] is to characterize equation solving in process algebra. In particular, consistency is used to prove satisfiability of a given specification.

In [3, 7] a study of the complexity of different decision problems for MTSs and Mixed transition systems is presented. In particular it is shown that thorough refinement for strong and weak semantics is EXPTIME-complete, considering that branching alphabet refinement is between these two is expected to have the same complexity but further study is necessary.

Hussain and Huth [48] also study the consistency problem, solving it for multiple 3-valued models, representing different views, with the same alphabet. But, they focus on the complexity of the relevant model-checking procedures: consistency, satisfiability, and validity. Instead, our work addresses the more general problem of supporting engineering activities in model elaboration. Finally, our models are more general than the models of Hussain and Huth in that we merge models with different vocabularies and τ transitions, but less general in that Hussian and Huth handle hybrid constraints, e.g., restricting the number of states a given proposition is evaluated in.

MTSs are defined over flat state spaces: Δ^r and Δ^p give a partial description of the behaviours over a *finite* set of states. Huth et al. [50] use the mixed power-domain of Gunter [39] to generalize MTSs to non-flat state spaces, modelled as domains. This extension is more expressive than MTSs, and can be used to represent other formalisms such as Mixed MTSs or partial Kripke structures. This extension guarantees uniqueness of merge, but at the expense of a non-trivial consistency check for one model. Checking whether a model has at least one valid implementation cannot be done in polynomial time. This complexity is "transferred" to the modeller when he or she attempts to understand a model drawing an intuition from the implementation set given by that model. In addition, non-uniqueness of merge over MTSs encountered in our work can be seen as an opportunity for elicitation, validation, and negotiation of partial descriptions.

Other approaches support merging inconsistent and incomplete views, i.e., enabling reasoning in the presence of inconsistencies [27, 79]. In [27] it is assumed that only states with the same label can be merged, and a similar consistency assumption is made in [95] in the context of UML differencing. On the other hand, in [79] a more general category-theoretic approach is presented which is based on the observation that it is not always clear how to relate two views. They use graph morphisms to express such relationships, enabling the user to provide this as a third argument to merge. Nejati and Chechik present a framework for merging 4-valued Kripke structures [73], where the fourth value indicates disagreement. The aim is to support negotiation for inconsistency resolution, helping users identify and prioritize disagreements through visualization. A key difference with the above approaches is that we focus on merging models that describe only the observable behaviour of a system. Hence, simulation-like relations, as opposed to relations that focus on the state structure, are appropriate for merging. Models merged by [27, 79, 73, 76] include state information, and consequently other notions of preservation, such as isomorphism, apply.

An alternative to partial operational descriptions, which we focus on, is the use of declarative specifications. For instance, classical logics are partial in that a theory denotes a set of models, hence they support merging as the conjunction of theories which denotes the intersection of their models. Similarly, Live Sequence Charts [41] support merging through logical conjunction, as each chart can be interpreted as a temporal logic formula. We believe that our approach is complementary and the fact that it models explicitly possible but not required behaviour may facilitate exploration and validation of unknown behaviours facilitating further elicitation.

The operation of merging also arises in several other related areas, including synthesis of StateChart models from scenarios [61], program integration [46], and combining program summaries for software model-checking [5].

The notion of system composition through partial descriptions is at the core of approaches to feature interaction in telecommunication systems (e.g., [12, 75]). These approaches aim to describe a product through a composition of features.

When features are described via operational models such as state machines, the formalisms require that each feature be fully specified. It is not possible to model the fact that certain aspects of a feature are presently unknown, to compose these features without having to resolve the unknowns, and to analyze the resulting model in the presence of these unknowns. Thus, there is no support for reasoning about a family of products resulting from the unknown aspects of the features used to build the product model. Furthermore, the notions of merge and composition, prevalent in the feature interaction literature, differ from the ones used in this work (see [74] for details).

Abstraction and Property Preservation

Explicit partiality corresponds naturally to the lack of information at modelling time. Our work has focused on finding a more elaborate model, based on refinement, that preserves the properties of two consistent partial models. The reverse of this process is abstraction, in which a less refined model is constructed. Unlike merge, abstract models are usually hidden from the user for use in automatic procedures, e.g., for efficient model-checking of large or infinite state systems. In addition, the notion of consistency is irrelevant in abstraction, as there is always a model that refines an abstraction, namely, the original model itself. However, like merge, soundness of abstractions with respect to property preservation is of fundamental importance in order for abstractions to be of any use when checking properties.

The approach of extending transition systems with a second transition relation describing unknown behaviour was originally proposed by [66], and independently by [22]. Larsen and Thomsen introduced MTSs as a solution to the completeness limitation of LTSs, and proved that Hennessy-Milner logic [43] characterizes strong refinement. Dams' Mixed Transition Systems [22, 21], which are MTSs that do not assume that all required transitions are possible transitions, are used for abstracting Kripke structures. It is shown that 3-valued CTL* properties are preserved by the refinement preorder between these models [22]. Bruns and Godefroid introduced partial Kripke Structures (PKs) [11], which have a single unlabelled transition relation and 3-valued state propositions. They show that 3-valued CTL defined over PKs characterizes their completeness preorder.

In [50] Huth et al. introduced Kripke MTSs (KMTSs) – a state-based version of MTSs. A KMTS has two transition relations, as in an MTS, but instead of having labelled transitions, each state is labelled with a set of 3-valued propositions. It is shown that 3-valued μ-calculus characterizes refinement defined over KMTSs, which is used as the basis for a 3-valued framework for program analysis. A proof that model checking of 3-valued μ-calculus properties for KMTS can be reduced to model checking of 3-valued μ-calculus properties on regular Kripke Transition Systems is provided in [49].

When a property evaluates to *maybe* in an abstract model, the model must be further refined (where refinement corresponds to splitting abstract states). [83] shows that even standard methods of refining abstract models (e.g. [37]) are not monotonic with respect to property preservation. Shoham and Grumberg define Generalized KMTSs (GKMTSs), an extension of KMTSs with hyper-transitions, as a solution to this problem, and obtain a monotonic abstraction-refinement framework with respect to 3-valued CTL.

Finally, MTSs, KMTSs, and PKs have the same expressive power [38]. The same is true for 4-valued Kripke structure, Mixed Transition Systems and Generalized

Kripke MTSs [40, 94], which represent another group of equivalent models.

9.2 Evaluation of Achievements

The motivation for the work presented in this thesis comes from the need to better support the elaboration of partial behaviour models in the context of currently used software development practices. In particular, we aimed to develop theoretical results and practical tools to support modelling processes that allow both the level of detail of the models to be increased over time as well as different viewpoints of the system to be integrated into the overall system description as they become available. These two characteristics are essential if an MDSE approach is to be adopted as part of industry standard software developments techniques, which are characterised by their incremental and iterative nature.

We have achieved these goals by developing a novel MTS semantics, branching alphabet semantics, which allows for the elaboration of model behaviour with increasing level of detail. We have shown that, unlike other MTS semantics, this semantics is sound with respect to alphabet extension and can therefore support iterative software development practices, as desired. Moreover, we have developed a merge algorithm that successfully allows for different views of the same system to be combined, further enhancing its applicability in the context of currently used software development techniques. We have also studied relevant properties of merge, providing essential results to support the use of compositional modelling from a practical perspective. The development of a software tool that allows the user to apply the different concepts presented in this thesis and analyse the results further contributes to this aim.

Additional contributions of our work include a thorough analysis of the strengths and shortcomings from a software elaboration point of view of the different previously existing MTS semantics. As part of this analysis we solved open theoretical questions providing results such as a characterisation of consistency, a complete merge algorithm, and a proof that MTSs are not closed with respect to the merge operation. This analysis also led us to the development of branching alphabet semantics, which we have shown combines the benefits while avoiding the main limitations of existing MTS semantics, as shown in our case study.

9.3 Future work

In the future, we intend to continue experimentation by conducting larger case studies in order to further explore the opportunities and limitations of the work presented in this thesis.

The fact that MTSs are not closed under merge, i.e., that MCRs may not exist, prompts the question of whether other partial behaviour modelling formalisms could be developed to better support incremental behaviour model elaboration. In the case of MTSs, we expect to address the practical difficulties introduced by merging MTS models with no least common refinement by developing an n-ary merge operator that constructs a common refinement from an unbounded number of MTSs and iteratively abstracts the result. Such an operator would remove the necessity of choosing MCRs for the $n - 1$ pairwise merges needed to merge n MTSs and would prevent the propagation of any incompleteness introduced by merging models, further facilitating the elaboration process.

Bibliography

[1] M. Abadi and L. Lamport. "The Existence of Refinement Mappings". *Theoretical Computer Science*, 82(2):253–284, 1991.

[2] Adam Antonik, Michael Huth, Kim Larsen, Ulrik Nyman, and Andrzej Wasowski. 20 Years of Mixed and Modal Specifications. June 2008.

[3] Adam Antonik, Michael Huth, Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. Complexity of decision problems for mixed and modal specifications. In *FoSSaCS*, 2008.

[4] Ralph-Johan Back and Joakim von Wright. Trace refinement of action systems. In *Proceedings of the Concurrency Theory*, CONCUR '94, pages 367–384, London, UK, 1994. Springer-Verlag.

[5] T. Ball, V. Levin, and F. Xie. "Automatic Creation of Environment Models via Training". In *Proceedings of 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 93–107. Springer, 2004.

[6] Sebastian S. Bauer, Line Juhl, Kim G. Larsen, Axel Legay, and Jiri Srba. Extending modal transition systems with structured labels. *Mathematical Structures in Computer Science*, 2012. To appear.

[7] Nikola Benes, Jan Kretínský, Kim Guldstrand Larsen, and Jirí Srba. Checking thorough refinement on modal transition systems is EXPTIME-Complete. In Leucker and Morgan [67], pages 112–126.

[8] Jean Bézivin. On the unification power of models. *Software and System Modeling*, 4(2):171–188, 2005.

[9] B. Boem and R. Turner. *Balancing Agility and Discipline: A Guide for the Perplexed.* Person Education, 2004.

[10] Greg Brunet. "A Characterization of Merging Partial Behavioural Models". Master's thesis, Univ. of Toronto, January 2006.

[11] G. Bruns and P. Godefroid. "Model Checking Partial State Spaces with 3-Valued Temporal Logics". In *CAV'99*, volume 1633 of *LNCS*, pages 274–287, 1999.

[12] Muffy Calder and Evan H. Magill, editors. *Feature Interactions in Telecommunications and Software Systems VI, May 17-19, 2000, Glasgow, Scotland, UK.* IOS Press, 2000.

[13] Marsha Chechik, Greg Brunet, Dario Fischbein, and Sebastián Uchitel. Partial behavioural models for requirements and early design. In Ed Brinksma, David Harel, Angelika Mader, Perdita Stevens, and Roel Wieringa, editors, *MMOSS*, volume 06351 of *Dagstuhl Seminar Proceedings.* Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[14] Marsha Chechik, Benet Devereux, Steve M. Easterbrook, and Arie Gurfinkel. Multi-valued symbolic model-checking. *ACM Trans. Softw. Eng. Methodol.*, 12(4):371–408, 2003.

[15] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. “Progress on the State Explosion Problem in Model Checking”. In R. Wilhelm, editor, *Informatics. 10 Years Back. 10 Years Ahead*, volume 2000 of *LNCS*, pages 176–194. Springer-Verlag, 2001.

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

[17] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, 1996.

[18] Rance Cleaveland and Matthew Hennessy. Testing equivalence as a bisimulation equivalence. *Formal Asp. Comput.*, 5(1):1–20, 1993.

[19] CREWS. Cooperative Requirements Engineering With Scenarios. `http://Sunsite.Informatik.RWTH-Aachen.DE/CREWS`, 1999.

[20] Jim Cunningham and Anthony Finkelstein. “Formal Requirements Specification: the FOREST Project”. In *Proceedings of 3rd International Workshop on Software Specification and Design*, pages 186–192. IEEE CS Press, 1986.

[21] D. Dams, R. Gerth, and O. Grumberg. “Abstract Interpretation of Reactive Systems”. *ACM Transactions on Programming Languages and Systems*, 2(19):253–291, 1997.

[22] Dennis Dams. *Abstract Interpretation and Partition Refinement for Model Checking.* PhD thesis, Eindhoven University of Technology, The Netherlands, July 1996.

[23] R. Diaz-Redondo, J. Pazos-Arias, and A. Fernandez-Vilas. “Reusing Verification Information of Incomplete Specifications”. In *Proceedings of the 5th Workshop on Component-Based Software Engineering*, 2002.

[24] Nicolás D'Ippolito, Dario Fischbein, Marsha Chechik, and Sebastián Uchitel. Mtsa: The modal transition system analyser. In *ASE*, pages 475–476. IEEE, 2008.

[25] Nicolás D'Ippolito, Dario Fischbein, Howard Foster, and Sebastián Uchitel. Mtsa: Eclipse support for modal transition systems construction, analysis and elaboration. In Li-Te Cheng, Alessandro Orso, and Martin P. Robillard, editors, *ETX*, pages 6–10. ACM, 2007.

[26] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. "Property Specification Patterns for Finite-state Verification". In *Proceedings of 2nd Workshop on Formal Methods in Software Practice*, March 1998.

[27] S. Easterbrook and M. Chechik. "A Framework for Multi-Valued Reasoning over Inconsistent Viewpoints". In *Proceedings of International Conference on Software Engineering (ICSE'01)*, pages 411–420, Toronto, Canada, May 2001. IEEE Computer Society Press.

[28] Harald Fecher and Heiko Schmidt. Comparing disjunctive modal transition systems with an one-selecting variant. *J. Log. Algebr. Program.*, 77(1-2):20–39, 2008.

[29] Dario Fischbein. `http://www.doc.ic.ac.uk/~fdario/Thesis/CaseStudy/`.

[30] Dario Fischbein, Víctor A. Braberman, and Sebastián Uchitel. A sound observational semantics for modal transition systems. In Leucker and Morgan [67], pages 215–230.

[31] Dario Fischbein, Greg Brunet, Nicolas D'Ippolito, Marsha Chechik, and Sebastian Uchitel. Weak alphabet merging of partial behaviour models. *ACM Transactions on Software Engineering and Methodology*, 21(2), 2011.

[32] Dario Fischbein and Sebastián Uchitel. Behavioural model elaboration using mts. In *"Copenhagen" Meeting on Modal Transition Systems*, 2007.

[33] Dario Fischbein and Sebastián Uchitel. On correct and complete strong merging of partial behaviour models. In Mary Jean Harrold and Gail C. Murphy, editors, *SIGSOFT FSE*, pages 297–307. ACM, 2008.

[34] Dario Fischbein, Sebastián Uchitel, and Víctor A. Braberman. A foundation for behavioural conformance in software product line architectures. In Robert M. Hierons and Henry Muccini, editors, *ROSATEA*, pages 39–48. ACM, 2006.

[35] Melvin Fitting. "Many-Valued Modal Logics". *Fundamenta Informaticae*, 15(3-4):335–350, 1991.

[36] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. Prentice Hall, 1991.

[37] P. Godefroid, M. Huth, and R. Jagadeesan. "Abstraction-based Model Checking using Modal Transition Systems". In K.G. Larsen and M. Nielsen, editors, *Proceedings of 12th International Conference on Concurrency Theory (CONCUR'01)*, volume 2154 of *LNCS*, pages 426–440, Aalborg, Denmark, 2001. Springer.

[38] P. Godefroid and R. Jagadeesan. "On the Expressiveness of 3-Valued Models". In *Proceedings of 4th International Conference on Verification, Model*

Checking, and Abstract Interpretation (VMCAI'03), volume 2575 of *LNCS*, pages 206–222. Springer, January 2003.

[39] C. Gunter. "The Mixed Powerdomain". *Theoretical Computer Science*, 103(2):311–334, 1992.

[40] A. Gurfinkel, O. Wei, and M. Chechik. "Systematic Construction of Abstractions for Model-Checking". In *Proceedings of 7th International Conference on Verification, Model-Checking, and Abstract Interpretation (VMCAI'06)*, volume 3855 of *LNCS*, pages 381–397, Charleston, SC, January 2006. Springer.

[41] D. Harel, H. Kugler, and A. Pnueli. "Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements.". In *Formal Methods in Software and Systems Modeling*, pages 309–324, 2005.

[42] Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In J. W. de Bakker and Jan van Leeuwen, editors, *ICALP*, volume 85 of *Lecture Notes in Computer Science*, pages 299–309. Springer, 1980.

[43] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, 1985.

[44] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[45] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, New York, 1985.

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

[47] A. Hunter and B. Nuseibeh. "Managing Inconsistent Specifications: Reasoning, Analysis and Action". *ACM Transactions on Software Engineering and Methodology*, 7(4):335–367, October 1998.

[48] Altaf Hussain and Michael Huth. "On Model Checking Multiple Hybrid Views". In *Proceedings of 1st International Symposium on Leveraging Applications of Formal Methods*, pages 235–242, November 2004.

[49] M. Huth. "Model-Checking Modal Transition Systems Using Kripke Structures". In *Proceedings of Third International Workshop on Verification, Model-Checking, and Abstract Interpretation*, Venice, Italy, January 2002.

[50] M. Huth, R. Jagadeesan, and D. Schmidt. "A Domain Equation for Refinement of Partial Systems". Submitted to Mathematical Structures in Computer Science, 2002.

[51] M. Huth, R. Jagadeesan, and D. A. Schmidt. "Modal Transition Systems: A Foundation for Three-Valued Program Analysis". In *Proceedings of 10th European Symposium on Programming (ESOP'01)*, volume 2028 of *LNCS*, pages 155–169. Springer, 2001.

[52] Hans Hüttel and Kim Guldstrand Larsen. The use of static constructs in a modal process logic. In Albert R. Meyer and Michael A. Taitslin, editors, *Logic at Botik*, volume 363 of *Lecture Notes in Computer Science*, pages 163–180. Springer, 1989.

[53] IEEE. “IEEE Recommended Practice for Software Requirements Specifications Standard 830”. Technical Standard 830, Wallace S. Read (Chair), 1994.

[54] ITU-T. “ITU-T Recommendation Z.120: Message Sequence Chart (MSC)”. *ITU-T*, 1993.

[55] Michael Jackson. *Software requirements & specifications: a lexicon of practice, principles and prejudices.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[56] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[57] Zoltán Juhász, Ádám Sipos, and Zoltán Porkoláb. Implementation of a finite state machine with active libraries in c++. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *GTTSE*, volume 5235 of *Lecture Notes in Computer Science*, pages 474–488. Springer, 2007.

[58] Robert M. Keller. Formal verification of parallel programs. *Commun. ACM*, 19(7):371–384, 1976.

[59] J. Kramer, J. Magee, and M. Sloman. “CONIC: an Integrated Approach to Distributed Computer Control Systems”. *IEE Proceedings*, 130(1):1–10, 1983.

[60] Jeff Kramer and Jeff Magee. *Concurrency: State Models & Java Programs, 2nd Edition.* Worldwide Series in Computer Science. John Wiley Sons, April 2006.

[61] I. Krueger, R. Grosu, P. Scholz, and M. Broy. "From MSCs to Statecharts". In Franz J. Rammig, editor, *Distributed and Parallel Embedded Systems.* Kluwer Academic Publishers, 1999.

[62] K. Larsen, B. Steffen, and C. Weise. "The Methodology of Modal Constraints". In *Formal Systems Specification*, volume 1169 of *LNCS*, pages 405–435. Springer, 1996.

[63] K. Larsen and L. Xinxin. "Equation Solving Using Modal Transition Systems". In *5th Annual IEEE Symposium on Logic in Computer Science*, pages 108–117, 1990.

[64] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. On modal refinement and consistency. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR*, volume 4703 of *Lecture Notes in Computer Science*, pages 105–119. Springer, 2007.

[65] Kim Guldstrand Larsen, Bernhard Steffen, and Carsten Weise. A constraint oriented proof methodology based on modal transition systems. In Ed Brinksma, Rance Cleaveland, Kim Guldstrand Larsen, Tiziana Margaria, and Bernhard Steffen, editors, *TACAS*, volume 1019 of *Lecture Notes in Computer Science*, pages 17–40. Springer, 1995.

[66] Kim Guldstrand Larsen and Bent Thomsen. A modal process logic. In *LICS*, pages 203–210. IEEE Computer Society, 1988.

[67] Martin Leucker and Carroll Morgan, editors. *Theoretical Aspects of Computing - ICTAC 2009, 6th International Colloquium, Kuala Lumpur, Malaysia, August 16-20, 2009. Proceedings*, volume 5684 of *Lecture Notes in Computer Science*. Springer, 2009.

[68] Gavin Lowe and A. W. Roscoe. Using csp to detect errors in the tmn protocol. *IEEE Trans. Software Eng.*, 23(10):659–669, 1997.

[69] Jeff Magee, Jeff Kramer, and Dimitra Giannakopoulou. Analysing the behaviour of distributed software architectures: a case study. In *FTDCS*, pages 240–247. IEEE Computer Society, 1997.

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

[71] Robin Milner. A modal characterisation of observable machine-behaviour. In *CAAP '81: Proceedings of the 6th Colloquium on Trees in Algebra and Programming*, pages 25–34, London, UK, 1981. Springer-Verlag.

[72] Robin Milner. Handbook of theoretical computer science (vol. b). chapter Operational and algebraic semantics of concurrent processes, pages 1201–1242. MIT Press, Cambridge, MA, USA, 1990.

[73] S. Nejati and M. Chechik. "Let's Agree to Disagree". In *Proceedings of 20th IEEE International Conference on Automated Software Engineering (ASE'05)*, pages 287 – 290. IEEE Computer Society, 2005.

[74] S. Nejati and M. Chechik. "Behavioural Model Fusion: Experiences from Two Telecommunication Case Studies". In *Proceedings of ICSE'08 Workshop on Modeling in Software Engineering (MiSE'08)*, May 2008.

[75] S. Nejati, M. Chechik, M. Sabetzadeh, S. Uchitel, and P. Zave. "Towards Compositional Synthesis of Evolving Systems". In *Proceedings of SIGSOFT Conference on Foundations of Software Engineering (FSE'08)*, pages 285–296, November 2008.

[76] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. "Matching and Merging of Statecharts Specifications". In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pages 54–64, 2007.

[77] D. M.R. Park. Concurrency and automata on infinite sequences. Technical report, Coventry, UK, UK, 1981.

[78] Alexander Moshe Rabinovich. Checking equivalences between concurrent systems of finite agents (extended abstract). In Werner Kuich, editor, *ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 696–707. Springer, 1992.

[79] M. Sabetzadeh and S.M. Easterbrook. "Analysis of Inconsistency in Graph-Based Viewpoints: A Category-Theoretic Approach". In *Proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 12–21. IEEE Computer Society, October 2003.

[80] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–, February 2006.

[81] Steve Schneider. Security properties and csp. In *IEEE Symposium on Security and Privacy*, pages 174–187. IEEE Computer Society, 1996.

[82] Steve Schneider and S. A. Schneider. *Concurrent and Real Time Systems: The CSP Approach.* John Wiley & Sons, Inc., New York, NY, USA, 1999.

[83] S. Shoham and O. Grumberg. "Monotonic Abstraction-Refinement for CTL". In *Proceedings of 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 546–560. Springer-Verlag, April 2004.

[84] German Sibay, Sebastián Uchitel, and Víctor A. Braberman. Existential live sequence charts revisited. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *ICSE*, pages 41–50. ACM, 2008.

[85] S. Uchitel, J. Kramer, and J. Magee. "Behaviour Model Elaboration using Partial Labelled Transition Systems". In *ESEC/FSE'03*, pages 19–27, 2003.

[86] S. Uchitel, J. Kramer, and J. Magee. "Synthesis of Behavioural Models from Scenarios". *IEEE Transactions on Software Engineering*, 29(2):99–115, 2003.

[87] S. Uchitel, J. Kramer, and J. Magee. "Incremental Elaboration of Scenario-Based Specifications and Behaviour Models using Implied Scenarios". *ACM Transactions on Software Engineering and Methodology*, 13(1):37–85, 2004.

[88] Sebastián Uchitel, Greg Brunet, and Marsha Chechik. Behaviour model synthesis from properties and scenarios. In *ICSE*, pages 34–43. IEEE Computer Society, 2007.

[89] Sebastián Uchitel, Greg Brunet, and Marsha Chechik. Synthesis of partial behavior models from properties and scenarios. *IEEE Trans. Software Eng.*, 35(3):384–406, 2009.

[90] Sebastián Uchitel and Marsha Chechik. Merging partial behavioural models. In Richard N. Taylor and Matthew B. Dwyer, editors, *SIGSOFT FSE*, pages 43–52. ACM, 2004.

[91] Rob J. van Gabbeek and W. Peter Weijland. Branching time and abstraction in bisimulation semantics. *J. ACM*, 43(3):555–600, 1996.

[92] Rob J. van Glabbeek. What is branching time semantics and why to use it? pages 469–479, 2001.

[93] Rob J. van Glabbeek and W. Peter Weijland. Branching time and abstraction in bisimulation semantics. *J. ACM*, 43:555–600, May 1996.

[94] O. Wei, A. Gurfinkel, and M. Chechik. "Mixed Transition Systems Revisited". In *Proceedings of 10th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'09)*, volume 5403 of *LNCS*, pages 349–365, January 2009.

[95] Z. Xing and E. Stroulia. "UMLDiff: An Algorithm for Object-Oriented Design Differencing". In *Proceedings of 20th IEEE International Conference on Automated Software Engineering (ASE'05)*, pages 54–65. IEEE Computer Society, 2005.