Weak Alphabet Merging of Partial Behaviour Models

DARIO FISCHBEIN, GREG BRUNET, NICOLAS D'IPPOLITO, MARSHA CHECHIK, SEBASTIAN UCHITEL

Constructing comprehensive operational models of intended system behaviour is a complex and costly task, which can be mitigated by the construction of partial behaviour models, providing early feedback and subsequently elaborating them iteratively. However, how should partial behaviour models with different viewpoints covering different aspects of behaviour be composed? How should partial models of component instances of the same type be put together? In this paper, we propose *model merging* of Modal Transition Systems (MTSs) as a solution to these questions. MTS models are a natural extension of Labelled Transition Systems that support explicit modelling of what is currently unknown about system behaviour. We formally define model merging based on weak alphabet refinement, which guarantees property preservation, and show that merging consistent models is a process that should result in a minimal common weak alphabet refinement (MCR). In this paper, we provide theoretical results and algorithms that support such a process. Finally, because in practice MTS merging is likely to be combined with other operations over MTSs such as parallel composition, we also study the *algebraic* properties of merging and apply these, together with the algorithms that support MTS merging, in a case study.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Temporal Logic*

General Terms: Design

Additional Key Words and Phrases: MTS, Merge, Partial Behaviour Models.

1. INTRODUCTION

Behaviour modelling and analysis has been shown to be successful in uncovering subtle design errors [Clarke and Wing, 1996]. However, the adoption of such technologies by practitioners has been slow. Partly, this is due to the difficulty of constructing behaviour models – this task requires considerable expertise in modelling notations that developers often lack.

Automated synthesis techniques have been studied to aid the construction and elaboration of behaviour models. In particular, synthesis from scenario-based specifications such

The first and the third authors are at the Department of Computing, Imperial College, 180 Queen's Gate, London, SW7 2RH, UK and can be reached at fdario@gmail.com and srdipi@gmail.com. The second author is now at Oracle, Inc. in California, but the work was done while he was in the Department of Computer Science, University of Toronto, Toronto, ON M5S 2E4, Canada. He can be reached at greg.brunet@utoronto.ca. The fourth author is at the University of Toronto as well and can be reached at chechik@cs.toronto.edu. The fifth author is at Imperial College and at the University of Buenos Aires, Argentina. He can be reached at s.uchitel@doc.ic.ac.uk.

as message sequence charts [ITU-T, 1993] and goal models [van Lamsweerde, 2004] are increasingly popular (e.g. [Uchitel et al., 2005; Dupont et al., 2008]). Such specifications tend to be of a partial nature, and the automated construction of behaviour models is expected to support elicitation of scenarios and goals in the context of an elicit-model-validate cycle.

We have shown that classical, two valued, behaviour models such as labelled transition systems (LTSs) [Keller, 1976] are inadequate to support such iterative elaboration as they cannot capture the partial information provided by heterogeneous specifications that contain both existential and universal statements of system behaviour [Uchitel et al., 2009].

When supporting the incremental elaboration of partial specifications of system behaviour, a more appropriate type of model to synthesize is one in which currently unknown aspects of behaviour can be explicitly modelled [Uchitel et al., 2009]. These models 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. Behaviour models that distinguish between these kinds of behaviour are referred to as *partial behaviour models*. A number of such modeling formalisms exist, e.g., Partial Labelled Transition Systems (PLTSs) [Uchitel et al., 2003a], multi-valued state machines [Diaz-Redondo et al., 2002], Mixed Transition Systems [Dams, 1996], multi-valued Kripke structures [Fitting, 1991; Bruns and Godefroid, 1999; Chechik et al., 2003]), and Modal Transition Systems (MTSs) [Larsen and Thomsen, 1988], and promising results on their use to support incremental modelling and viewpoint analysis has been reported.

In this paper, we concentrate on using MTSs for which synthesis techniques for various specification language styles, such as Message Sequence Charts and Sequence Diagrams, Use Cases and Goal Models, have been developed [Sibay et al., 2008; Uchitel et al., 2009].

The semantics of a partial behaviour model can be thought of as a set of traditional behaviour models. For instance, MTS semantics can be given in terms of sets of LTSs that provide all of the behaviour required by the MTS, do not provide any of the behaviour prohibited by the MTS, and make different decisions on whether or not to provide the MTS's unknown behaviour.

The notions of strong and weak refinement [Larsen et al., 1996] between MTSs capture this intuition formally and provide an elegant way of describing the process of behaviour model elaboration as one in which behaviour information is acquired and introduced into the behaviour model incrementally, gradually refining a given MTS until it characterizes a single LTS.

LTSs can be thought of as partial models if a notion of refinement, such as *trace inclusion* and *simulation* [Milner, 1989], is adopted. For instance, if we interpret the behavior explicitly described in an LTS as required and all other behavior as "yet to be determined", an LTS that simulates another can be interpreted as a partial model in which some of the "yet to be determined" behaviour has been identified as required. This interpretation of LTSs can be thought of as providing a *lower bound* to the final, complete, description of the system behavior, since the latter must provide at least the required behavior, while perhaps implementing additional behavior. This view is taken by approaches that construct LTS models from scenario-based specifications, e.g., [Krueger et al., 1999].

An alternative interpretation of LTSs is to consider the explicitly described behaviour as possible, but not yet confirmed, while the behaviour not described as forbidden. As

more information becomes available, the possible behaviour can be pruned by making it prohibited. This interpretation considers the described behavior as an *upper bound* of the behaviour of the final system. This view is taken by approaches that synthesize LTSs from safety properties (e.g., [Letier et al., 2008]), where the synthesized model describes all of the behaviour that does not violate known properties.

However, partial behaviour models such as MTSs can describe *both* an upper and a lower bound to the intended system behaviour, allowing both bounds to be refined simultaneously. As more information becomes available, unknown or unclassified behaviour gets changed into either required or prohibited behaviour. MTSs come equipped with two sets of transitions: required, which provide a lower bound to system behavior, and possible, which provide an upper bound.

A particularly useful notion in the context of software and requirements engineering is that of *merge* [Larsen and Thomsen, 1988; Uchitel and Chechik, 2004]. Merging of operational behaviour models is similar to conjunction of declarative descriptions. The LTSs described by a merge are those that provide all the required behaviour and that do not provide any of the prohibited behaviour of the MTSs being merged. In other words, merging attempts to build a new MTS that represents the intersection of the sets of LTSs described by models being merged.

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 [Huth et al., 2001; Larsen and Thomsen, 1988; Larsen et al., 1996; Larsen et al., 1995; Fischbein and Uchitel, 2008; Uchitel et al., 2007; Uchitel et al., 2009]. However, these studies make the strong assumption that alphabets of these models are the same. Hence, existing MTS semantics, *strong* and *weak* [Larsen et al., 1996], require MTSs to have the same alphabet.

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. 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. In particular, a semantics that allows for alphabet refinement supports merging various partial behaviour models with different alphabets and hence of diverse scopes.

In this paper, we present a study of Modal Transition Systems under a new semantics, called *weak alphabet semantics*, which supports alphabet refinement. We also present results and algorithms that support the elaboration of partial behaviour models. The paper makes a number of contributions.

The *first contribution* of this paper is a novel refinement notion called *weak alphabet refinement*. Not only does it capture the elaboration process in which behaviour is incrementally identified as required or prohibited, as in strong and weak semantics, but it also enables augmenting the scope of the description as novel relevant concepts are identified. We further show that this refinement preserves properties expressed in fluent linear temporal logic (FLTL).

The *second contribution* of this paper is a study of consistency under weak and weak alphabet refinement. Two models are said to be *consistent* if a common refinement exists. Consistency is a precondition for computing merge, as a minimal common refinement cannot be built if there are no common refinements. We define a notion of a consistency relation which is a complete characterization of consistency for MTSs under weak refinement

and a sufficient condition for consistency under weak alphabet refinement. This contribution extends the current state of the art: until now, consistency has only been characterized under strong semantics [Fischbein and Uchitel, 2008], and a sufficient condition for it has been provided under weak semantics [Larsen et al., 1996].

The *third contribution* of this paper is a set of automated methods for constructing common refinements and merge. Given two models for which a consistency relation exists, we provide an operator that constructs their common refinement and an algorithm that builds their merge – the least common refinement, if it exists, or a set of minimal common refinements, otherwise.

The *fourth contribution* of this paper is a study of the algebraic properties of merge and parallel composition and their relationship with refinement. We provide results that are essential to support compositional construction of system behaviour models. Such construction includes both merging partial behaviour models of *the same component* and parallel composition of partial behaviour models of *different components* which communicate to provide the system-level functionality. We exemplify the utility of some of the algebraic properties, theoretical results and algorithms presented in the paper by applying them to support behaviour model elaboration within a Mine Pump case study.

The rest of the paper is organized as follows. In Section 3, we give preliminary definitions used throughout the paper, as well as introduce 3-valued linear temporal logic of fluents. Section 4 describes merging MTSs. In Section 5, we present a discussion on consistency. In Section 6, we give algorithms for constructing common refinements and merging MTSs. In Section 7, we present positive and negative results on algebraic properties for merging, while providing insights into the implications that these results have on engineering partial behaviour models. In Section 8, we briefly comment on the tool support that we have developed for computing MTS refinement and merging. In Section 9, we provide a case study that illustrates the utility of our theoretical results. Finally, Section 10 presents a summary of our results, compares them with related approaches, and discusses directions for future work. Proofs of selected theorems are given in the Appendix.

2. MOTIVATING EXAMPLE

In this section, we provide a small example which motivates the work presented in this paper.

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". 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. 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.

No other behaviour is allowed, i.e., cash withdrawal, topping up or exiting are not allowed in states 0 or 1.

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. The model $\mathcal{B}$ in Figure 1 depicts an MTS with $N = 2$. Note that the property does not prescribe the 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 [ITU-T, 1993], use-case diagrams [Jacobson, 2004] and structured declarative specifications such as [Dwyer et al., 1998]. MTS synthesis techniques have been studied [Uchitel et al., 2007; Uchitel et al., 2009] but are beyond the scope of this paper. 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 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 semantics of models being composed? How to treat models with different languages? In this paper, we answer these questions.

Furthermore, if a model of the assumptions made on the user behaviour (model $\mathcal{D}$ in Figure 1) were provided, how can we reason about the emergent behaviour of the user and the partially described ATM (model $\mathcal{C}$)? Are there ATM implementations conforming to $\mathcal{C}$ which can produce deadlocking situations? Do non-deadlocking implementations preserve the intended behaviour of the ATM? In this paper we study parallel composition of partial behaviour models and present results that answer these questions.

3. BACKGROUND

In this section, we provide the necessary definitions, fix the notation, and introduce a new 3-valued variant of the linear temporal logic of fluents (FLTL) for reasoning about MTSs. Specifically, Section 3.1 discusses LTSs and their extension to MTSs, and Section 3.2 reviews relations and operations on MTSs: strong and weak refinement, hiding, and parallel composition. Finally, Section 3.3 defines 3-valued FLTL and provides refinement preservation and model checking results.

3.1 Transition Systems

We begin with the familiar concept of labelled transition systems (LTSs) [Keller, 1976], which are widely used for modelling and analyzing the behaviour of concurrent and distributed systems. An LTS is a state transition system where transitions are labelled with actions. The set of actions of an LTS is called its *communicating alphabet* and constitutes the interactions that the modelled system can have with its environment. In addition, LTSs can have transitions labelled with τ, representing actions that are not observable by the

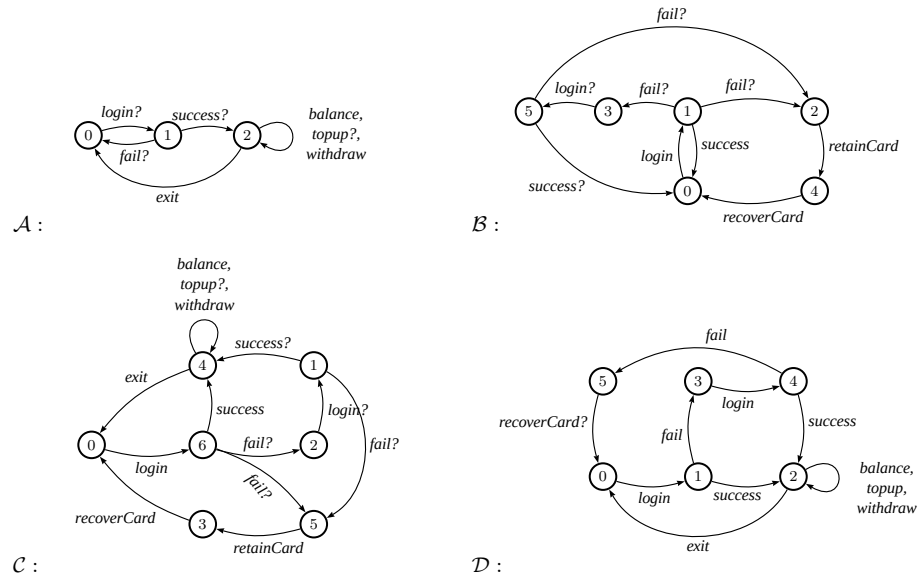


Fig. 1. MTS and LTS models for an ATM.

environment. Models $\mathcal{E}$ and $\mathcal{F}$ in Figure 3 are example LTSs. Recall that states labelled by 0 represent initial states. Transitions labelled with sets are abbreviations for an individual transition on every action in the set, and the fonts $\mathcal{M}$, $\mathbb{M}$, and M are used for naming specific transition systems.

In the following definitions, we use *States* to denote the universal set of states, *Act* – the universal set of observable action labels, τ – the non-observable action and $Act_\tau = Act \cup \{\tau\}$.

DEFINITION 3.1. (Labelled Transition System) *A* Labelled Transition System *(LTS) is a tuple $L = (S, A, \Delta, s_0)$, where $S \subseteq States$ is a finite set of states, $A \subseteq Act_\tau$ is a set of actions (labels), $\Delta \subseteq (S \times A \times S)$ is a transition relation between states, and $s_0 \in S$ is the initial state.*

DEFINITION 3.2. *$L = (S, A, \Delta, s_0)$ be an LTS. The* communicating alphabet *of L (denoted αL) is $A \setminus \{\tau\}$.*

Modal Transition Systems (MTSs) [Larsen and Thomsen, 1988] allow explicit modelling of what is *not* known about the behaviour of a system. They extend LTSs with an additional set of transitions that model the interactions with the environment that the system cannot be guaranteed to provide, but equally cannot be guaranteed to prohibit.

DEFINITION 3.3. (Modal Transition System) *A* Modal Transition System *(MTS) M is a structure $(S, A, \Delta^r, \Delta^p, s_0)$, where $\Delta^r \subseteq \Delta^p$, (S, A, Δ^r, s_0) is an LTS representing* required *transitions of the system and (S, A, Δ^p, s_0) is an LTS representing* possible *transitions of the system.*

We use $\wp$ to denote the universe of all MTSs.

Figures 1 and 3 shows a graphical representation of MTSs that model ATMs. For example, $\mathcal{G}$ models those ATMs in which the top-up feature may or may not be present.

Transitions labelled with a question mark are those in $\Delta^p \setminus \Delta^r$. LTSs are a special type of MTSs in which the sets of possible and required transitions coincide; thus, models $\mathcal{E}$ and $\mathcal{F}$ can be considered as LTSs or MTSs.

DEFINITION 3.4. (MTS Transitions) *Given an MTS $M = (S, A, \Delta^r, \Delta^p, s_0)$ and an action $\ell \in A$, we say that*

—*M has a* required *transition on ℓ (denoted $M \xrightarrow{\ell}_r M'$) 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 $M \xrightarrow{\ell}_p M'$) iff $(s_0, \ell, s'_0) \in \Delta^p$ 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., $\forall s'_0 \in S \cdot (s_0, \ell, s'_0) \notin \Delta^p$.*

For example, in MTS $\mathcal{G}$ in Figure 3, there is a required transition between states 0 and 1 ($\mathcal{G}_0 \xrightarrow{\texttt{login}}_r \mathcal{G}_1$ and also a possible transition between these states $\mathcal{G}_0 \xrightarrow{\texttt{login}}_\mathrm{p} \mathcal{G}_1$, since $\Delta^r \subseteq \Delta^p$), a possible but not required self-loop in state 3 ($\mathcal{G}_3 \xrightarrow{\texttt{topup}}_\mathrm{p} \mathcal{G}_3$) and no transition on `topup` from state 0 ($\mathcal{G}_0 \overset{\texttt{topup}}{\not\longrightarrow}$).

DEFINITION 3.5. (Initial State) *For an MTS $M = (S, A, \Delta^r, \Delta^p, s_0)$ and a state $n \in S$, we denote changing the initial state of M from s_0 to n as M_n.*

DEFINITION 3.6. *Let $M = (S, A, \Delta^r, \Delta^p, s_0)$ be an MTS. The* communicating alphabet *of M (denoted αM) is $A \setminus \{\tau\}$.*

Allowing MTSs to have different communicating alphabets enables us to provide descriptions with different scopes. For instance, the communicating alphabets of $\mathcal{A}$ and $\mathcal{B}$ differ, allowing for more compact descriptions. For example, the user operations provided by an ATM once a user is logged in, in $\mathcal{A}$, can be described independently, and thus more compactly, of the procedure for recovering cards that have been retained due to too many successive failed login attempts, in model $\mathcal{B}$.

Our treatment of alphabets is in line with the process algebra semantics such as FSP (Finite State Processes) [Magee and Kramer, 1999]. Unless stated otherwise, we assume that the communicating alphabet of an MTS coincides with the set of observable actions for which the MTS has a transition. However, this is not necessarily the case: an MTS may not include transitions labelled with an action from its communicating alphabet, meaning that this action is prohibited from occurring in all states.

Finally, we define the shared alphabet of two MTSs.

DEFINITION 3.7. (Shared Alphabet) *We call the set $\alpha M \cap \alpha N$ the* shared *alphabet of MTSs M and N, and $(\alpha M \setminus \alpha N) \cup (\alpha N \setminus \alpha M)$ the* non-shared *alphabet of M and N.*

3.2 Relations and Operations on MTSs

Refinement can be seen as being a "more defined than" relation between two partial models. Intuitively, refinement in MTSs is about converting transitions which are possible but not required into required transitions or removing them altogether: an MTS N refines an MTS M if N preserves all of the required and all of the prohibited behaviours of M.

Alternatively, N refines M if N can simulate the required behaviour of M, and M can simulate the possible behaviour of N. Refinement captures the notion of elaborating a partial description by iteratively adding more information about the required and prohibited behaviour of the system to be.

Larsen [Larsen et al., 1996] introduced notions of strong and weak refinement for MTSs. We reproduce these definitions below, making explicit that they only apply to MTSs with identical communicating alphabets.

DEFINITION 3.8. (Strong Refinement) *Let M and N be MTSs such that $\alpha M = \alpha N$. N is a* refinement *of M, written $M \preceq_{\mathrm{s}} N$, iff (M, N) is contained in some* strong refinement relation $R \subseteq \wp \times \wp$*, for which the following holds for all $\ell \in Act_\tau$ and for all $(M', N') \in R$:*

$$1.\ \forall M'' \cdot (M' \xrightarrow{\ell}_r M'' \Rightarrow \exists N'' \cdot N' \xrightarrow{\ell}_r N'' \wedge (M'', N'') \in R)$$
$$2.\ \forall N'' \cdot (N' \xrightarrow{\ell}_p N'' \Rightarrow \exists M'' \cdot M' \xrightarrow{\ell}_p M'' \wedge (M'', N'') \in R)$$

The above definition is given in terms of possible and required transitions and the set of possible transitions is a superset of the required ones. Hence, $M \preceq_{\mathrm{s}} N$ if the required behaviour of M is required in N and any behaviour which is possible in N is possible in M. That is, N can "convert" possible but not required behaviour of M into required or prohibited behaviour but may not introduce "new" such behaviour.

The MTS $\mathcal{C}$ in Figure 1 is refined by the LTS $\mathcal{G}$ of Figure 3 ($\mathcal{C} \preceq \mathcal{G}$). The additional information in $\mathcal{G}$ is that the ATM cannot retain a card after a first attempt and must allow a second attempt at logging in. Intuitively, some possible transitions have been dropped (those that model whether the ATM retains a card after a failed first attempt at logging in) and some required transitions have been added (those related to retaining cards after two failed logins). The refinement relation between these models is

$$R = \{(\mathcal{C}_0, \mathcal{G}_0), (\mathcal{C}_6, \mathcal{G}_1), (\mathcal{C}_2, \mathcal{G}_2), (\mathcal{C}_1, \mathcal{G}_4), (\mathcal{C}_4, \mathcal{G}_3), (\mathcal{C}_5, \mathcal{G}_5), (\mathcal{C}_3, \mathcal{G}_6)\}.$$

When two models refine each other, we say that they are *equivalent*:

DEFINITION 3.9. (Equivalence) *Let M and N be MTSs. M is* strongly equivalent *to N (denoted $M \equiv_s N$) if and only if $M \preceq N$ and $N \preceq M$.*

When restricted to LTSs, strong equivalence is bisimulation [Milner, 1989].

Although (strong) refinement captures the notion of model elaboration, it requires the alphabets of the processes being compared to be the same. In practice, model elaboration can lead to augmenting the alphabet of the system model to describe behaviour aspects that previously had not been taken into account. To capture this aspect of model elaboration, we use two concepts: hiding and weak refinement.

Hiding is 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 3.10. (Hiding) *Let $M = (S, A, \Delta^r, \Delta^p, s_0)$ be an MTS and $X \subseteq Act$ be a set of actions. M with the actions of X hidden, denoted $M \backslash X$, is an MTS $(S, A \backslash X, \Delta^{r'}, \Delta^{p'}, s_0)$, where $\Delta^{r'}$ and $\Delta^{p'}$ are the smallest relations that satisfy the rules below, where $\ell \in Act_\tau$.*

$$\frac{M \xrightarrow{\ell}_\gamma M'}{(M \backslash X) \xrightarrow{\ell}_\gamma (M' \backslash X)}\ \ell \notin X,\ \gamma \in \{r,p\} \qquad \frac{M \xrightarrow{\ell}_\gamma M'}{(M \backslash X) \xrightarrow{\tau}_\gamma (M' \backslash X)}\ \ell \in X,\ \gamma \in \{r,p\}$$

For a set $Y \subseteq Act$, we use $M@Y$ to denote $M\backslash(Act\backslash Y)$.

DEFINITION 3.11. (Notation for Transitions) *Let $w = l_1, \ldots, l_k$ be a word over Act_τ^* and let M be an MTS. 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 $\gamma \in \{r, p\}$, $M \overset{\ell}{\Longrightarrow}_\gamma M'$ denotes $M(\xrightarrow{\tau}_\gamma)^*(\xrightarrow{\ell}_\gamma)(\xrightarrow{\tau}_\gamma)^* M'$. Similarly, $M \overset{\hat{\ell}}{\Longrightarrow}_\gamma M'$ 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 $\Longrightarrow_\gamma$).*

For example, for consider model $\mathcal{A}$ in Figure 1 and define $\mathcal{A}' = \mathcal{A}\backslash\{\texttt{success}\}$. For $\mathcal{A}'$, ${\mathcal{A}'}_0 \overset{\texttt{login fail}}{\longrightarrow}_\text{p} {\mathcal{A}'}_0$ and ${\mathcal{A}'}_0 \xrightarrow{\hat{\tau}}_\text{p} {\mathcal{A}'}_0$. In addition, ${\mathcal{A}'}_0 \overset{\hat{\texttt{login}}}{\longrightarrow}_\text{p} {\mathcal{A}'}_1$, ${\mathcal{A}'}_1 \xrightarrow{\hat{\tau}}_\text{p} {\mathcal{A}'}_2$, ${\mathcal{A}'}_2 \overset{\hat{\texttt{exit}}}{\longrightarrow}_\text{p} {\mathcal{A}'}_0$, and thus ${\mathcal{A}'}_0 \overset{\hat{\texttt{login}}}{\Longrightarrow}_\text{p} {\mathcal{A}'}_2$ and ${\mathcal{A}'}_1 \overset{\hat{\texttt{exit}}}{\Longrightarrow}_\text{p} {\mathcal{A}'}_0$.

DEFINITION 3.12. (Traces) *For an MTS $M = (S, A, \Delta^r, \Delta^p, s_0)$, a* trace $\pi = \ell_0,\ell_1,\ldots$ *where $\ell_i \in Act$ is a* required trace *in M iff there exists an M' such that $M \overset{\pi}{\Longrightarrow}_\text{r} M'$. Similarly, π is a* possible trace *in M iff there exists an M' such that $M \overset{\pi}{\Longrightarrow}_\text{p} M'$.*

DEFINITION 3.13. (Infinite Traces) *We denote the set of infinite required and possible traces of an MTS M by* REQTR*(M) and* POSTR*(M) respectively. For an LTS $L = (S, A, \Delta, s_0)$, we denote by* TR*(L) the set of (infinite) required traces of its embedding into an MTS $M = (S, A, \Delta, \Delta, s_0)$, so that* TR*$(L)$* = REQTR*$(M)$.*

We use infinite traces to give semantics to FLTL properties in Section 3.3 below.

In order to compare models that have unobservable actions, possibly generated through hiding, we need an alternative notion of refinement, called *weak refinement*, which ignores differences related to τ-transitions. Weak equivalence of MTSs can be defined in the same manner as strong equivalence.

DEFINITION 3.14. (Weak Refinement) *Let MTSs N and M such that $\alpha M = \alpha N$ be given. N is a* weak refinement *of M, written $M \preceq_\text{w} N$, iff (M, N) is contained in some* weak refinement relation *$R \subseteq \wp \times \wp$, for which the following holds for all $\ell \in Act_\tau$ and for all $(M', N') \in R$:*

$$1.\ \forall M'' \cdot (M' \xrightarrow{\ell}_r M'' \Rightarrow \exists N'' \cdot N' \overset{\hat{\ell}}{\Longrightarrow}_r N'' \wedge (M'', N'') \in R)$$
$$2.\ \forall N'' \cdot (N' \xrightarrow{\ell}_p N'' \Rightarrow \exists M'' \cdot M' \overset{\hat{\ell}}{\Longrightarrow}_p M'' \wedge (M'', N'') \in R)$$

Consider again the MTS $\mathcal{C}$ shown in Figure 1. It captures the requirements of models $\mathcal{A}$ and $\mathcal{B}$ (as claimed in Section 2) because it weakly refines (with appropriate hiding) these models.

If actions in set $X = \{\texttt{retainCard}, \texttt{recoverCard}\}$ are hidden in $\mathcal{C}$, then $\mathcal{C}' = \mathcal{C} \setminus X$ weakly refines $\mathcal{A}$ ($\mathcal{A} \preceq_\text{w} \mathcal{C}'$) via the relation

$$R = \{(\mathcal{A}_0, \mathcal{C}'_0), (\mathcal{A}_1, \mathcal{C}'_6), (\mathcal{A}_1, \mathcal{C}'_1), (\mathcal{A}_2, \mathcal{C}'_4), (\mathcal{A}_0, \mathcal{C}'_5), (\mathcal{A}_0, \mathcal{C}'_3), (\mathcal{A}_0, \mathcal{C}'_2)\}.$$

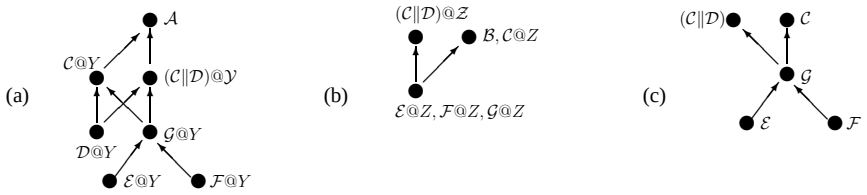


Fig. 2. Weak Refinements between ATM models for alphabets (a) Y; (b) Z; (c) $Y \cup Z$.

The weak refinement relation between $\mathcal{C}'' = \mathcal{C} \setminus \{\texttt{balance}, \texttt{withdraw}, \texttt{topup}, \texttt{exit}\}$ and $\mathcal{B}$ is

$$R = \{(\mathcal{B}_0, \mathcal{C}''_0), (\mathcal{B}_1, \mathcal{C}''_6), (\mathcal{B}_2, \mathcal{C}''_5), (\mathcal{B}_3, \mathcal{C}''_2), (\mathcal{B}_4, \mathcal{C}''_3), (\mathcal{B}_5, \mathcal{C}''_1), (\mathcal{B}_0, \mathcal{C}''_4)\}.$$

Figure 2 depicts weak refinements that hold between models discussed above. Figure 2(a) relates models with the alphabet

$$Y = \{\texttt{login}, \texttt{success}, \texttt{fail}, \texttt{exit}, \texttt{balance}, \texttt{topup}, \texttt{withdraw}\}.$$

Figure 2(b) relates models with alphabets restricted to

$$Z = \{\texttt{login}, \texttt{success}, \texttt{fail}, \texttt{retainCard}, \texttt{recoverCard}\},$$

and Figure 2(c) relates models with the alphabet $Y \cup Z$. Nodes with multiple labels indicate models that are weakly equivalent.

LTSs that refine an MTS M are complete descriptions of the system behaviour and thus are called *implementations* of M. The semantics of an MTS M can be thought of as a model that represents the set of LTSs that implement it.

DEFINITION 3.15. (Implementation and Implementation Relation) *An LTS L is an* implementation *of an MTS M if and only if L is a refinement of M ($M \preceq L$). We denote the set of implementations of M as $\mathcal{I}(M)$ and refer to the refinement relation between an MTS M and an LTS $L \in \mathcal{I}(M)$ as an* implementation relation.

The LTSs $\mathcal{E}\setminus\{\texttt{balance}, \texttt{withdraw}, \texttt{topup}, \texttt{exit}\}$ and $\mathcal{F}\setminus\{\texttt{balance}, \texttt{withdraw}, \texttt{topup}, \texttt{exit}\}$ are both weak implementations of $\mathcal{B}$. However, $\mathcal{B}$ also admits weak implementations that model ATMs which retain cards after the first failed attempt to login.

An implementation is *deadlock free* if all states have outgoing transitions. We refer to the set of deadlock-free implementations of M as $\mathcal{I}_\infty(M)$. Deadlock-free implementations are also parameterized by their refinement type (e.g., strong and weak).

DEFINITION 3.16. (Deadlock-free Implementation) *An LTS $L = (S_L, A, \Delta_L, s_{0L})$ is a* deadlock-free *implementation of an MTS M if and only if $M \preceq L$ and for all $s \in S_L$, there exists $a \in A$ and $s' \in S_L$ such that $L_s \xrightarrow{a} L_{s'}$.*

In the remainder of this paper, we assume *weak* interpretations of the notions of (deadlock-free) implementations and equivalence, unless stated otherwise.

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

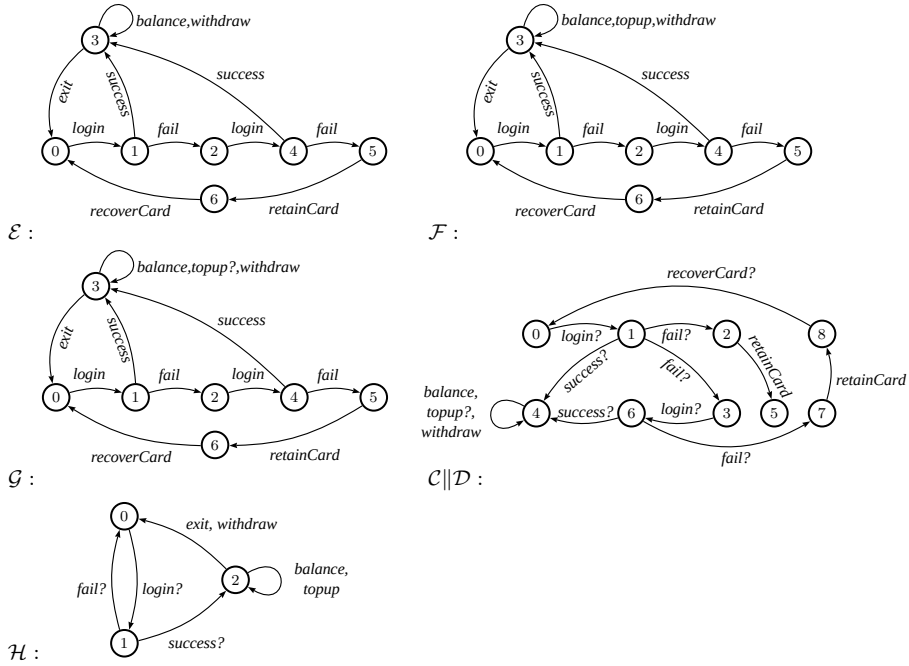


Fig. 3. Additional MTS and LTS models for an ATM. Models $\mathcal{C}$ and $\mathcal{D}$ are shown in Figure 1.

DEFINITION 3.17. (Determinism) *Let $M = (S, A, \Delta^r, \Delta^p, s_0)$ be an MTS. M is* deterministic *iff $\tau \notin A$ and*

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

Larsen and Thomsen [Larsen and Thomsen, 1988] define a parallel composition operator over MTSs, intended to describe how models of two different systems work together:

DEFINITION 3.18. (Parallel Composition) *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 $\ell \in Act_\tau$:*

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

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

When restricted to LTSs, the parallel composition operator defined above becomes the standard one (e.g., [Magee and Kramer, 1999]).

In the rules in Definition 3.18, "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).

Model $\mathcal{C}||\mathcal{D}$ in Figure 3 depicts the parallel composition of models $\mathcal{C}$ (of the ATM) and $\mathcal{D}$ (of the user). The resulting MTS has a deadlock 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 the login, tries to login again (see state 2) and yet the ATM does not allow it, instead attempting to retain the card (see state 5). 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 3.1. *Parallel composition satisfies the following properties [Hüttel and Larsen, 1989]:*

1.	*(Commutativity)*	$M\|N$	$=$	$N\|M$.
2.	*(Associativity)*	$(M\|N)\|P$	$=$	$M\|(N\|P)$.
3.	*(Strong Monotonicity)*	$M \preceq_s N$	$\Rightarrow$	$M\|P \preceq_s N\|P$.
4.	*(Weak Monotonicity)*	$M \preceq_w N$	$\Rightarrow$	$M\|P \preceq_w N\|P$.

3.3 3-valued FLTL

In this paper, we describe properties using Fluent Linear Temporal Logic (FLTL) [Giannakopoulou and Magee, 2003]. Linear temporal logics (LTL) [Pnueli, 1977] are widely used to describe behaviour requirements [Giannakopoulou and Magee, 2003; van Lamsweerde and Letier, 2000; Kazhamiakin et al., 2004]. The motivation for choosing an LTL of fluents is that it provides a uniform framework for specifying and model-checking state-based temporal properties in event-based models [Giannakopoulou and Magee, 2003]. An LTL formula checked against an LTS model requires interpreting propositions as the occurrence of events in the LTS model. Some properties can be rather cumbersome to express as sequences of events, while describing them in terms of states is simpler. Fluents provide a way of defining abstract states that can be checked on an LTS. In this section, we review the 3-valued Kleene logic [Kleene, 1952], 3-valued variant of Fluent Linear Temporal Logic (FLTL) [Uchitel et al., 2009], and results for the property preservation of refinement.

3.3.1 *3-Valued Logic.* The truth values $\mathbf{t}$ (*true*), $\mathbf{f}$ (*false*), and $\perp$ (*maybe*, *unknown*) form the Kleene logic, which we refer to as $\mathbf{3}$. These truth values can have two orderings, $\sqsubseteq$ (truth), which satisfies $\mathbf{f} \sqsubseteq \perp \sqsubseteq \mathbf{t}$, and $\sqsubseteq_{inf}$ (information), which satisfies $\perp \sqsubseteq_{inf} \mathbf{t}$ and $\perp \sqsubseteq_{inf} \mathbf{f}$ (i.e., *maybe* gives the least amount of information). With respect to the truth ordering, the values $\mathbf{t}$ and $\mathbf{f}$ behave classically for $\wedge$ (and), $\vee$ (or), and $\neg$ (negation). The following identities hold for $\perp$:

$$\perp \wedge \mathbf{t} = \perp \quad \perp \wedge \mathbf{f} = \mathbf{f} \quad \perp \vee \mathbf{t} = \mathbf{t} \quad \perp \vee \mathbf{f} = \perp \quad \neg\perp = \perp.$$

3.3.2 *Fluent LTL.* FLTL [Giannakopoulou and Magee, 2003] is a linear-time temporal logic for reasoning about fluents. A *fluent Fl* is defined by an initial value $Initially_{Fl}$ and a pair of sets I_{Fl} (the set of initiating actions) and T_{Fl} (the set of terminating actions).

$Fl = \langle I_{Fl}, T_{Fl} \rangle_{Initially_{Fl}}$ where $I_{Fl}, T_{Fl} \subseteq Act$ and $I_{Fl} \cap T_{Fl} = \emptyset$ and $Initially_{Fl}$ is *true* or *false*

When omitted, the initial value of a fluent is assumed to be *false* . Every action $a \in Act$ induces a fluent, namely, a means $\langle a, Act \setminus \{a\} \rangle$.

$$
\begin{array}{ll}
\pi \models \mathit{Fl} & \triangleq\ \pi^0 \models \mathit{Fl} \\
\pi \models \neg\varphi & \triangleq\ \neg(\pi \models \varphi) \\
\pi \models \varphi \vee \psi & \triangleq\ (\pi \models \varphi) \vee (\pi \models \psi) \\
\pi \models \varphi \wedge \psi & \triangleq\ (\pi \models \varphi) \wedge (\pi \models \psi) \\
\pi \models \mathbf{X}\varphi & \triangleq\ \pi^1 \models \varphi \\
\pi \models \varphi \mathbf{U} \psi & \triangleq\ \exists i \geq 0 \cdot \pi^i \models \psi \wedge \forall\, 0 \leq j < i \cdot \pi^j \models \varphi \\
\pi \models \varphi\ \mathbf{W}\ \psi & \triangleq\ \pi \models (\varphi\ \mathbf{U}\ \psi) \vee \Box\varphi \\
\pi \models \Diamond\varphi & \triangleq\ \pi \models \mathbf{t}\ \mathbf{U}\ \varphi \\
\pi \models \Box\varphi & \triangleq\ \pi \models \neg\Diamond\neg\varphi
\end{array}
$$

Fig. 4. Semantics of the satisfaction operator.

Given a set of fluents Φ, an FLTL formula is defined inductively using the standard boolean connectives and temporal operators $\mathbf{X}$ (next), $\mathbf{U}$ (strong until), $\mathbf{W}$ (weak until), $\Diamond$ (eventually), and $\Box$ (always), as follows:

$$\varphi ::= \mathit{Fl} \mid \neg\varphi \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \mathbf{X}\varphi \mid \varphi\mathbf{U}\psi \mid \varphi\mathbf{W}\psi \mid \Diamond\varphi \mid \Box\varphi,$$

where $\mathit{Fl} \in \Phi$.

Let Π be the set of infinite traces over *Act*. For a trace $\pi = a_0, a_1, \ldots \in \Pi$ and $i \in \mathbb{N}$, let π^i denote the part of π starting at position i. We say that π^i satisfies a fluent *Fl*, denoted $\pi^i \models \mathit{Fl}$, if and only if one of the following conditions holds:

- $\mathit{Initially}_{\mathit{Fl}} \wedge (\forall j \in \mathbb{N} \cdot 0 \leq j \leq i \Rightarrow a_j \notin T_{\mathit{Fl}})$
- $\exists j \in \mathbb{N} \cdot (j \leq i \wedge a_j \in I_f) \wedge (\forall k \in \mathbb{N} \cdot j < k \leq i \Rightarrow a_k \notin T_{\mathit{Fl}})$

In other words, a fluent holds at a time instant i if and only if it holds initially or some initiating action has occurred, but no terminating action has yet occurred. The interval over which a fluent holds is *closed* on the left and *open* on the right, since actions have an immediate effect on the value of fluents.

Given an infinite trace π, the satisfaction operator $\models$ is defined as shown in Figure 4. This definition is standard [Giannakopoulou and Magee, 2003] and yields a 2-valued operator.

In classical semantics, a formula $\varphi \in$ FLTL holds in a deadlock-free LTS L (denoted $L \models \varphi$) if it holds on every (infinite) trace produced by L. The 3-valued semantics of FLTL over an MTS M is given by the function $\| \cdot \|_t^M$ that, for each formula $\varphi \in$ FLTL, returns the truth value of φ in M, i.e., $\mathbf{t}$, $\mathbf{f}$ or $\bot$:

DEFINITION 3.19. (3-valued Semantics of FLTL ("thorough")) *Let φ be an FLTL property and M be an MTS s.t. $\mathcal{I}_\infty(M) \neq \emptyset$. φ evaluates to* true *in M iff it evaluates to* true *in all deadlock-free implementations of M. φ evaluates to* false *in M iff it evaluates to* false *in all deadlock-free implementations of M. φ evaluates to* maybe *in M iff it is* true *in some deadlock-free implementations of M and* false *in others. Formally, the function $\| \cdot \|_t^M$: FLTL $\longrightarrow \mathbf{3}$ is defined as follows:*

$$
\begin{array}{l}
\|\varphi\|_t^M = \mathbf{t} \quad \textit{iff}\ \forall L \in \mathcal{I}_\infty(M) \cdot L \models \varphi \\
\|\varphi\|_t^M = \mathbf{f} \quad \textit{iff}\ \forall L \in \mathcal{I}_\infty(M) \cdot L \not\models \varphi \\
\|\varphi\|_t^M = \bot \quad \textit{iff}\ \exists L, L' \in \mathcal{I}_\infty(M) \cdot L \models \varphi\ \wedge\ L' \not\models \varphi
\end{array}
$$

Definition 3.19 is similar to the thorough semantics given by [Bruns and Godefroid, 2000] but restricted only to non-deadlocking implementations. When M is a deadlock-free LTS,

the 3-valued semantics in Definition 3.19 reduces to the standard 2-valued semantics of FLTL given in Figure 4.

Refinement preserves all *true* and *false* FLTL properties under thorough semantics, e.g., the value of FLTL properties in more refined models increases w.r.t. the information ordering (can go from *maybe* to *true* or *false* but not the other way around).

THEOREM 3.1. (Preservation of FLTL) *[Uchitel et al., 2009] Let M and N be MTSs such that $M \preceq_w N$. Then, $\forall \varphi \in \text{FLTL} \cdot \|\varphi\|_t^M \sqsubseteq_{inf} \|\varphi\|_t^N$.*

For instance, consider the property $\Phi = \Box(\texttt{LoggedOut} \Rightarrow (\neg\texttt{balance} \wedge \neg\texttt{withdraw}))$ which states that withdrawals and balance requests must not occur when the user is logged out. This property holds for $\mathcal{A}$ and, since $\mathcal{A} \preceq_{\mathrm{w}} \mathcal{G}$, for $\mathcal{G}$ as well. Furthermore, Φ also holds for all refinements of $\mathcal{G}$, namely, the LTSs $\mathcal{E}$ and $\mathcal{F}$.

Given an MTS M with deadlock-free implementations, model-checking M against 3-valued FLTL formulas w.r.t. Definition 3.19 is likely as expensive as the procedure described in [Godefroid and Pitterman, 2009] (which is 2EXPTIME-complete in the size of the formula and polynomial in the size of the model), using a similar approach.

In practice, thorough semantics is often approximated by *compositional* or *inductive* [Wei et al., 2009] one defined as follows:

DEFINITION 3.20. (3-valued Semantics of FLTL (inductive)) *Let φ be an FLTL property and M be an MTS. The function $\|\cdot\|^M : \text{FLTL} \longrightarrow \mathbf{3}$ is defined as follows:*

$$\begin{array}{rcl} \|\varphi\|^M = \mathbf{t} & \triangleq & \forall \pi \in \textsc{PosTr}(M) \cdot \pi \models \varphi \\ \|\varphi\|^M = \mathbf{f} & \triangleq & (\exists \pi \in \textsc{ReqTr}(M) \cdot \pi \not\models \varphi) \vee \\ & & (\forall \pi \in \textsc{PosTr}(M) \cdot \pi \not\models \varphi) \\ \|\varphi\|^M = \bot & \triangleq & \neg(\|\varphi\|^M = \mathbf{t}) \wedge \neg(\|\varphi\|^M = \mathbf{f}) \end{array}$$

A formula φ is *true* in M (denoted $\|\varphi\|^M = \mathbf{t}$ or $M \models \varphi$), if every infinite trace in $\textsc{PosTr}(M)$ satisfies φ. A formula φ is *false* in M (denoted $\|\varphi\|^M = \mathbf{f}$ or $M \not\models \varphi$) if there is an infinite trace in $\textsc{ReqTr}(M)$ that refutes φ or if all infinite traces in $\textsc{PosTr}(M)$ refute φ. Otherwise, a formula φ evaluates to *maybe* in M (denoted $\|\varphi\|^M = \bot$).

Inductive semantics of FLTL approximates the thorough one, i.e., if a property is *true* (*false*) under inductive semantics, it is *true* (*false*) under thorough as well. Moreover, all *true* properties under thorough semantics are also *true* under inductive [Gurfinkel and Chechik, 2005]. However, some *maybe* properties under inductive semantics are in fact *false* under thorough:

THEOREM 3.2. Relationship between Inductive and Thorough Semantics of FLTL *Let M be an MTS and φ be an FLTL formula. Then,*

$$\begin{array}{l} \|\varphi\|^M = \mathbf{t} \Leftrightarrow \|\varphi\|_t^M = \mathbf{t} \\ \|\varphi\|^M = \mathbf{f} \Rightarrow \|\varphi\|_t^M = \mathbf{f} \end{array}$$

Inductive model-checking of an MTS M which has deadlock-free implementations against a 3-valued FLTL formula can be done using another standard procedure, described, e.g., in [Uchitel et al., 2009]. The procedure is based on creating *optimistic* and *pessimistic* versions of M, referred to as M^+ and M^-, respectively, and checking them using a classical model-checker, such as LTSA [Magee and Kramer, 1999]. Intuitively, M^+ is an LTS obtained from M by converting all maybe transitions to required and then removing those transitions which are not part of some infinite trace and all states that are not reachable

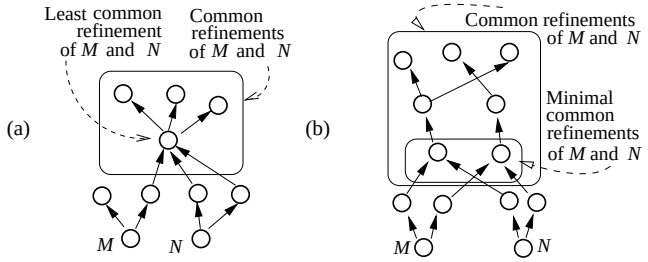


Fig. 5. Common refinements for consistent models M and N: (a) M and N have a least common refinement; (b) M and N have no least common refinement.

from the initial one. This procedure is linear in the size of the model and linear in the size of the formula [Bruns and Godefroid, 1999].

The above algorithm allows us to maintain the *monotonicity* property of the inductive semantics of FLTL under refinement, i.e., as in Theorem 3.1, in a more refined model, values for all properties increase w.r.t. the information ordering when compared to a less refined one.

THEOREM 3.3. (Monotonicity of Inductive Semantics of FLTL) *Let M and N be MTSs such that $M \preceq_w N$. Then, $\forall \varphi \in \text{FLTL} \cdot \|\varphi\|^M \sqsubseteq_{inf} \|\varphi\|^N$.*

In what follows, when we say "a property holds (fails) in a model", we mean inductive semantics of FLTL, unless explicitly mentioned otherwise.

4. MERGE

In this section, we introduce the notion of weak alphabet merging of modal transition systems. Section 4.1 argues for the notion of a common weak alphabet refinement as the basis for merge. In Section 4.2, we define merge of consistent models to be the least common (weak alphabet) refinement if it exists, and a minimal common refinement, otherwise.

Figure 5 provides an abstract summary of the concepts discussed in this section. In this figure, arrows depict weak alphabet refinements (i.e., an edge from P to Q indicates that P is weak alphabet refined by Q). For simplicity, we do not depict refinements that can be inferred by transitive closure of the ones depicted.

4.1 Common Weak Alphabet Refinement

The intuition we wish to capture by merging is that of augmenting the knowledge we have of the behaviour of a system by taking what we know from the two partial descriptions of the system. Clearly, the notion of refinement underlies this intuition as it captures the "more defined than" relation between two partial models. Hence, merging two models of the same system is about finding a common refinement for these models, i.e., finding a model that is more defined than both.

Models to be merged may have different scopes and hence different alphabets. Existing refinement relations for MTSs require models to have the same alphabet and consequently do not serve our purpose. In this section, we introduce weak alphabet refinement and discuss merging in this context.

Weak alphabet refinement allows comparing two models in which one has an alphabet that is a superset of the other. The refinement aims to capture the intuition of having more

information but only with respect to the common alphabet. It considers all other actions as out of scope for the comparison. Hence, weak alphabet refinement amounts to weak refinement in which actions in the extended alphabet are considered unobservable.

DEFINITION 4.1. (Weak Alphabet Refinement) *An MTS N is a* weak alphabet refinement *of an MTS M, written $M \preceq_{\mathrm{a}} N$, if $\alpha M \subseteq \alpha N$ and $M \preceq_{\mathrm{w}} N@\alpha M$.*

Note 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.

Like weak and strong refinement, weak alphabet refinement preserves FLTL. This follows from Theorem 3.1.

COROLLARY 4.1. (Preservation of FLTL) *Let M and N be MTSs such that $M \preceq_{\mathrm{a}} N$. Then, $\forall \varphi \in \text{FLTL} \cdot \|\varphi\|^M \sqsubseteq_{inf} \|\varphi\|^{N@\alpha M}$.*

The notion of common refinement is effectively parameterized by a particular refinement definition, e.g., strong, weak, and weak alphabet. In addition, we can use strong common refinement when models have the same vocabulary and do not use τ-transitions, weak common refinement when the models have the same vocabulary but do use τ-transitions, and weak alphabet common refinement if the alphabets are different. However, in this paper, we assume that common refinement refers to weak alphabet common refinement, unless otherwise specified.

DEFINITION 4.2. (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.

For example, $\mathcal{G}$ is a common refinement of $\mathcal{A}$ and $\mathcal{B}$. $\mathcal{G}$ 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. $\mathcal{G}$ refines $\mathcal{A}$ which describes operations to be provided by the ATM to users and also refines $\mathcal{B}$ which sets the maximum number of failed login attempts to two.

$\mathcal{G}$ 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 succesfull logins. Hence, a common refinement of $\mathcal{A}$ and $\mathcal{B}$, such as $\mathcal{G}$, requires that implementations allow for withdrawals.

In Figure 6, we depict the alphabet refinement relations that exist between the ATM models discussed previously.

4.2 Merge as a Minimal Common Refinement

Since $\mathcal{G}$ is a common refinement of $\mathcal{A}$ and $\mathcal{B}$, it preserves the required and prohibited behaviour of both models. However, a behaviour which is possible but not required in one model may become required or prohibited in the common refinement. For instance,

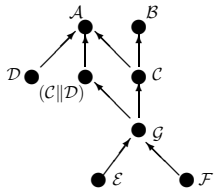


Fig. 6. Weak alphabet refinement relation between ATM models.

an unbounded number of consecutive failed logins is possible but not required in $\mathcal{A}$ but is disallowed in $\mathcal{G}$ (since otherwise it would not be a refinement of $\mathcal{B}$ as well).

However, $\mathcal{G}$ does introduce constraints on the behaviour of ATMs that are not specified in either $\mathcal{A}$ or $\mathcal{B}$, such as requiring an ATM to allow up to two failed logins and disallowing it to retain bank cards after the first failed attempt.

Model $\mathcal{C}$ is also a common refinement of $\mathcal{A}$ and $\mathcal{B}$, but unlike $\mathcal{G}$, it does not introduce additional constraints. Furthermore, $\mathcal{C}$ is the least refined MTS that is a common refinement of $\mathcal{A}$ and $\mathcal{B}$, and $\mathcal{G}$ is a refinement of $\mathcal{C}$.

Least common refinements are of interest because they compose two partial operational descriptions of the same system without introducing additional constraints on its behaviour.

DEFINITION 4.3. (Least Common Refinement) *Given a refinement notion, $\preceq$, an MTS P is a* least common refinement *(LCR) of modal transition systems M and N if P is a common refinement of M and N, $\alpha P = \alpha M \cup \alpha N$, and for any common refinement Q of M and N, $P \preceq Q$.*

As before, when referring to the least common refinement, we assume weak alphabet refinement, unless stated otherwise.

Assume that P_1 and P_2 are two LCRs of M and N, by Definition 4.3 it follows that $P_1 \preceq P_2$ and $P_2 \preceq P_1$ therefore $P_1 \equiv P_2$. This shows that least common refinements are unique up to observational equivalence, and hence we refer to *the* least common refinement, denoted $\mathcal{LCR}_{M,N}$ for models M and N.

An LCR of the original systems may not exist for two reasons. First, it is possible that no common refinement exists. Second, a common refinement may exist, but there may be no least one. We discuss these possibilities below.

Consider model $\mathcal{H}$ in Figure 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 the previous ATM models such as $\mathcal{A}$ or $\mathcal{G}$ which forbid logging in until an exit action has occurred. It is therefore impossible to build an ATM that satisfies both $\mathcal{H}$ and model $\mathcal{G}$.

DEFINITION 4.4. (Consistency) *Two MTSs M and N are* consistent *iff there exists an MTS P such that P is a common refinement of M and N.*

Now refer to the models shown in Figure 7(a). Models $\mathcal{I}$ and $\mathcal{J}$ do have common refinements, e.g., $\mathcal{K}$, $\mathcal{L}$ and $\mathcal{O}$, but no LCR. Intuitively, to find $\mathcal{LCR}_{\mathcal{I},\mathcal{J}}$, we must refine $\mathcal{I}$ into $\mathcal{I}'$ so that $(\mathcal{I}' \setminus \{\mathtt{a}, \mathtt{b}\})$ has a required transition on $\mathtt{c}$. Hence, we must transform the possible but not required transition on $\mathtt{c}$ in $\mathcal{I}$ to a required transition, and also transform one of the possible but not required transitions on $\mathtt{a}$ or $\mathtt{b}$. If we transform all three transitions, we obtain the model $\mathcal{O}$. However, if we choose not to transform the transition either on $\mathtt{a}$

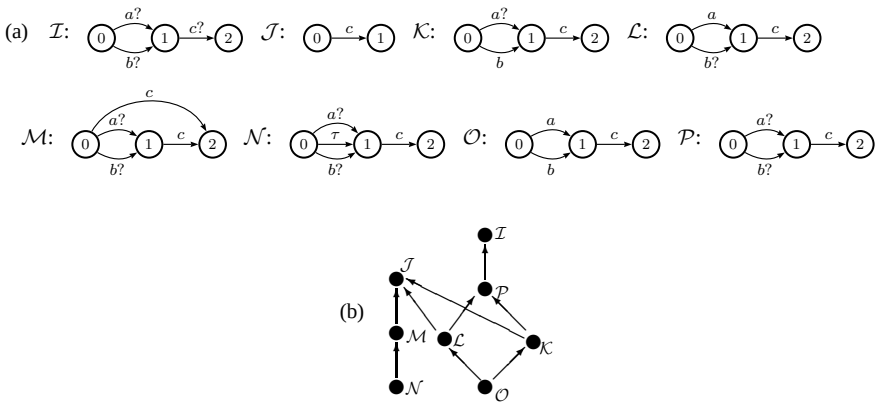


Fig. 7. (a) Example MTSs. (b) Weak alphabet refinement relation between models of (a).

or on b, then we obtain the models $\mathcal{K}$ and $\mathcal{L}$, which are both refined by, but not equivalent to $\mathcal{O}$. These common refinements are not comparable (neither is a refinement of the other) because of the different choices of which non-shared possible but not required transition is made required.

It is not possible to find common refinements of $\mathcal{I}$ and $\mathcal{J}$ which are less refined than $\mathcal{K}$ and $\mathcal{L}$. For example, $\mathcal{P}$ is less refined than both but is not a refinement of $\mathcal{J}$. Hence, we refer to $\mathcal{K}$ and $\mathcal{L}$ as the *minimal common refinements* of $\mathcal{I}$ and $\mathcal{J}$. Note that models $\mathcal{M}$ and $\mathcal{N}$ are incorrect attempts at building minimal common refinements of $\mathcal{I}$ and $\mathcal{J}$. These are not refinements of $\mathcal{I}$ because they can both transit on c from the initial state through (a sequence of) required transitions, whereas $\mathcal{I}$ cannot do so from its initial state.

DEFINITION 4.5. (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 $P \preceq Q$.*

Again, when referring to minimal common refinement, we assume weak alphabet refinement, unless stated otherwise.

We write $\mathcal{MCR}(M,N)$ to denote the set of MCRs of models M and N. By Definition 4.5 and Theorem 3.1, we have the following result.

COROLLARY 4.2. (Merge Preservation) *Let $P = P'@\alpha M$ with $P' \in \mathcal{MCR}(M,N)$, $Q = Q'@\alpha N$ with $Q' \in \mathcal{MCR}(M,N)$, s_M, s_N, s_P, s_Q the initial states of M, N, P, and Q, such that P and Q have deadlock-free implementations. Then:*

$$\forall \phi \in FLTL \cdot \quad (s_M \in [\![\phi]\!]^{\mathbf{t}} \Rightarrow s_P \in [\![\phi]\!]^{\mathbf{t}}) \wedge (s_M \in [\![\phi]\!]^{\mathbf{f}} \Rightarrow s_P \in [\![\phi]\!]^{\mathbf{f}}) \wedge$$
$$(s_N \in [\![\phi]\!]^{\mathbf{t}} \Rightarrow s_Q \in [\![\phi]\!]^{\mathbf{t}}) \wedge (s_N \in [\![\phi]\!]^{\mathbf{f}} \Rightarrow s_Q \in [\![\phi]\!]^{\mathbf{f}}).$$

That is, all *true* and *false* FLTL properties of M and N are preserved in model of $\mathcal{MCR}(M,N)$, when restricted to the appropriate alphabet, if the merge has deadlock-free implementations.

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 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

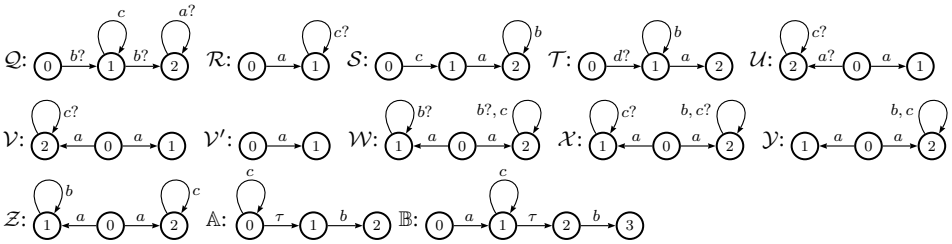


Fig. 8. Example MTSs.

choosing the most appropriate MCR. In Section 6.2, we present an algorithm that supports the merging process including the computation of multiple MCRs, should they exist.

5. CONSISTENCY

In this section, we define consistency relations and discuss the role of the largest such relation. Consistency relations are used in Section 6 to define two merge algorithms.

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 $N = (S_N, A_N, \Delta^r_N, \Delta^p_N, s_{0N})$, 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 (recall that 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 5.1. (Weak Consistency Relation) *A* weak consistency relation *is a binary relation $C \subseteq \wp \times \wp$, such that the following conditions hold for all $(M, N) \in C$, provided $\ell \in Act_\tau$:*

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

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 ($N \stackrel{\hat{\ell}}{\Longrightarrow}_p N'$) 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 5.2. (Weak Alphabet Consistency Relation) *A* weak alphabet consistency relation *is a binary relation $C \subseteq \wp \times \wp$, such that the following conditions hold for all*

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

1. $\forall M' \cdot (M \xrightarrow{\ell}_{\mathrm{r}} M' \ \wedge \ \ell \notin \alpha N \cup \{\tau\}) \Rightarrow \exists N' \cdot (N \stackrel{\hat{\tau}}{\Longrightarrow}_{\mathrm{p}} N' \ \wedge \ (M', N') \in C)$
2. $\forall M' \cdot (M \xrightarrow{\ell}_{\mathrm{r}} M' \ \wedge \ \ell \in \alpha N \cup \{\tau\}) \ \Rightarrow \exists x_1, \ldots, x_n \in (\alpha N \backslash \alpha M) \cdot$
   $\exists N_1, \ldots, N_n, N' \cdot (N \stackrel{x_1}{\Longrightarrow}_{\mathrm{p}} N_1 \cdots \stackrel{x_n}{\Longrightarrow}_{\mathrm{p}} N_n \stackrel{\hat{\ell}}{\Longrightarrow}_{\mathrm{p}} N') \wedge$
   $(\forall i \cdot 1 \leq i \leq n \Rightarrow (M, N_i) \in C) \ \wedge \ (M', N') \in C$
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 5.1): a behaviour required in one model must be possible in the other. However, it has two important differences. First, 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 5.1), then $N@\alpha M$ can simulate ℓ using, if necessary, a succession of possible transitions on actions not in αM. Second, 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 $\stackrel{x_i}{\Longrightarrow}_{\mathrm{p}}$ starting from N before the transition on ℓ (i.e., from N to N_x) must be consistent with M. This condition is similar to the one required for branching semantics for MTSs [Fischbein et al., 2006].

For example, consider the models $\mathbb{A}$ and $\mathbb{B}$ in Figure 8. These models are related by weak alphabet consistency:

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

The transition $\mathbb{A}_0 \xrightarrow{c}_r \mathbb{A}_0$ is simulated by $\mathbb{B}$ with $\mathbb{B}_0 \stackrel{ac}{\Longrightarrow}_r \mathbb{B}_1$. 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 5.1), 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 8, 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 $\mathcal{R}_0 \xrightarrow{a}_r \mathcal{R}_1$, $\mathcal{Q}$ must match this behaviour with $\mathcal{Q}_0 \stackrel{b}{\Longrightarrow}_{\mathrm{p}} \mathcal{Q}_1 \stackrel{b}{\Longrightarrow}_{\mathrm{p}} \mathcal{Q}_2 \stackrel{a}{\Longrightarrow}_{\mathrm{p}} \mathcal{Q}_2$. The definition of weak alphabet consistency requires intermediate state $\mathcal{Q}_1$ be related to state $\mathcal{R}_0$. But $\mathcal{Q}_1 \stackrel{c}{\Longrightarrow}_r \mathcal{Q}_1$ 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 $\mathcal{R}_0 \xrightarrow{a}_r \mathcal{R}_1$ with $\mathcal{Q}_0 \stackrel{a}{\Longrightarrow}_{\mathrm{p}} \mathcal{Q}_2$, 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 5.1. (Weak Consistency Relation Characterizes Weak Consistency) *Two MTSs are* weakly consistent *iff there is a weak consistency relation between them.*

THEOREM 5.2. (Weak Alphabet Consistency Relation Entails Weak Alphabet Consistency) *Two MTSs are* weakly alphabet consistent *if there is a weak alphabet consistency relation between them.*

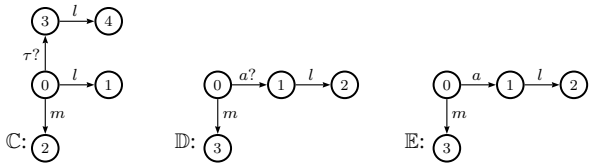


Fig. 9. MTSs for showing that weak alphabet consistency does not imply existence of a weak alphabet consistency relation between the models.

Note that the relationship in Theorem 5.2 (entailment) is weaker than the one in Theorem 5.1 (characterization). The converse of Theorem 5.2 does not hold. For example, consider models $\mathbb{C}$ and $\mathbb{D}$ in Figure 9. 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 $\mathbb{D}_1$.

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 (strong) consistency relation*. It is straightforward to show from Definition 5.2 that the union of two consistency relations is also a consistency relation.

DEFINITION 5.3. (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 8 defined over the vocabulary $\{\mathtt{a}, \mathtt{c}\}$. While $C_{\mathcal{RU}} = \{(\mathcal{R}_0, \mathcal{U}_0), (\mathcal{R}_1, \mathcal{U}_1), (\mathcal{R}_1, \mathcal{U}_2))\}$ is the largest consistency relation between them, $C'_{\mathcal{RU}} = C_{\mathcal{RU}} \setminus \{(\mathcal{R}_1, \mathcal{U}_2)\}$ 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 $\mathcal{V}'$. Unlike $\mathcal{V}'$, model $\mathcal{V}$ does not rule out the possibility of an action $\mathtt{c}$ occurring after an action $\mathtt{a}$ because $C_{\mathcal{RU}}$ 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 [Fischbein et al., 2006]. Such an algorithm (see Algorithm 5.1 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 5.1. WEAKALPHABETCONSISTENCYRELATION*(M, N)*

Input: *MTSs* $M = (S_M, A_M, \Delta^r_M, \Delta^p_M, s_{0M})$ *and* $N = (S_N, A_N, \Delta^r_N, \Delta^p_N, s_{0N})$
$C_0 = \{(M_s, N_t) \cdot s \in S_M$ *and* $t \in S_N\}$
Repeat
  $C_{i+1} \leftarrow \{(P, Q) \in C_i \mid (P, Q)$ *satisfies conditions 1-4 of Definition 5.2*$\}$
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, a fixpoint is reached in at most $\|C_0\|$ steps. Therefore, the time and the space complexity of this algorithm is $O(m \times n^4 \times log(n))$ and $O(n^2)$, respectively. 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 5.3. (Soundness of Algorithm 5.1) *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 5.1). However, since the converse of Theorem 5.2 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 5.4. (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 5.2. 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 that of consistency over their shared alphabet.

6. COMPUTING MERGE

In this section, we describe the algorithm for constructing merge under weak alphabet refinement. We first define the $+_{cr}$ operator 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 (Section 6.1). The result of $+_{cr}$ may not be an MCR in general. Hence, we present an algorithm, MERGE,

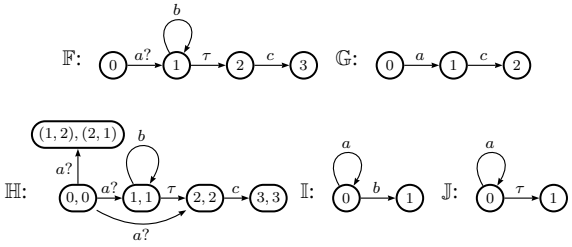


Fig. 10. Example MTSs for illustrating merge.

that iteratively abstracts $M +_{cr} N$ while guaranteeing that the result is still a common refinement of M and N. This algorithm copes with the case in which two models have more than one MCR.

6.1 Building a Common Refinement

In this subsection, we introduce the $+_{cr}$ operator 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 6.1. (The $+_{cr}$ operator) *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 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 $\ell \in Act_\tau$:*

RP $\dfrac{M \stackrel{\hat{\ell}}{\Longrightarrow}_r M',\ N \stackrel{\hat{\ell}}{\Longrightarrow}_{\mathrm{p}} N'}{(M,N) \stackrel{\ell}{\longrightarrow}_r (M',N')}$ **PR** $\dfrac{M \stackrel{\hat{\ell}}{\Longrightarrow}_{\mathrm{p}} M',\ N \stackrel{\hat{\ell}}{\Longrightarrow}_r N'}{(M,N) \stackrel{\ell}{\longrightarrow}_r (M',N')}$

PD $\dfrac{M \stackrel{\ell}{\Longrightarrow}_{\mathrm{p}} M',\ N \stackrel{\hat{\tau}}{\Longrightarrow}_{\mathrm{p}} N'}{(M,N) \stackrel{\ell}{\longrightarrow}_r (M',N')}\ \ell \notin (\alpha N \cup \{\tau\})$ **DP** $\dfrac{M \stackrel{\hat{\tau}}{\Longrightarrow}_{\mathrm{p}} M',\ N \stackrel{\ell}{\Longrightarrow}_{\mathrm{p}} N'}{(M,N) \stackrel{\ell}{\longrightarrow}_r (M',N')}\ \ell \notin (\alpha M \cup \{\tau\})$

PP $\dfrac{M \stackrel{\hat{\ell}}{\Longrightarrow}_{\mathrm{p}} M',\ N \stackrel{\hat{\ell}}{\Longrightarrow}_{\mathrm{p}} N'}{(M,N) \stackrel{\ell}{\longrightarrow}_{\mathrm{p}} (M',N')}$

Intuitively, the areas of agreement (described by the consistency relation) of the models being merged are traversed simultaneously, synchronizing on shared actions and producing transitions in the merged 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 6.1.

The cases in which the models agree on possible transition are handled by rule PP in Definition 6.1. 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. Our 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 10 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 $C_{\mathbb{IJ}} = \{(\mathbb{I}_0, \mathbb{J}_0), (\mathbb{I}_1, \mathbb{J}_1)\}$. $\mathbb{I} \xrightarrow{b}_r \mathbb{I}_1$, 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, $\mathbb{I}_0 +_{cr} \mathbb{J}_0 \not\xrightarrow{b} \mathbb{I}_1 +_{cr} \mathbb{J}_0$. However, $\mathbb{J}_0 \xrightarrow{\tau}_r \mathbb{J}_1$, 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 b, i.e., $\mathbb{I}_0 +_{cr} \mathbb{J}_0 \xrightarrow{b}_r \mathbb{I}_1 +_{cr} \mathbb{J}_1$. 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 transitions rather than possible transitions 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 7(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 10) 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 6.1. 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 6.1. ($+_{cr}$ builds CRs) *If there is a 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 10, 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 $\mathbb{F}_0 \overset{a}{\Longrightarrow}_{\mathrm{p}} \mathbb{F}_2$, $\mathbb{G}_0 \overset{a}{\Longrightarrow}_r \mathbb{G}_1$, and $(\mathbb{F}_2, \mathbb{G}_1) \in C_{\mathbb{FG}}$, it follows that $(\mathbb{F}_0, \mathbb{G}_0) \xrightarrow{a}_r (\mathbb{F}_2, \mathbb{G}_1)$ is a transition of $\mathbb{F} +_{cr} \mathbb{G}$ by the PR rule. Since $\mathbb{F}_2 \overset{c}{\Longrightarrow}_r \mathbb{F}_3$, $\mathbb{G}_1 \overset{c}{\Longrightarrow}_r \mathbb{G}_2$, and $(\mathbb{F}_3, \mathbb{G}_2) \in C_{\mathbb{FG}}$, it follows that $(\mathbb{F}_2, \mathbb{G}_1) \xrightarrow{c}_r (\mathbb{F}_3, \mathbb{G}_2)$ is a transition in $\mathbb{F} +_{cr} \mathbb{G}$. Hence, $\mathbb{F} +_{cr} \mathbb{G} = \mathbb{G}$, as desired.

6.2 The MERGE Algorithm

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 7, $\mathcal{I} +_{cr} \mathcal{J} = \mathcal{O}$, but MCRs of $\mathcal{I}$ and $\mathcal{J}$ are $\mathcal{K}$ and $\mathcal{L}$ (see the discussion in Section 4). 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 algorithm aimed to detect the required transitions resulting from the conservative rules that define 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 M and N.

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

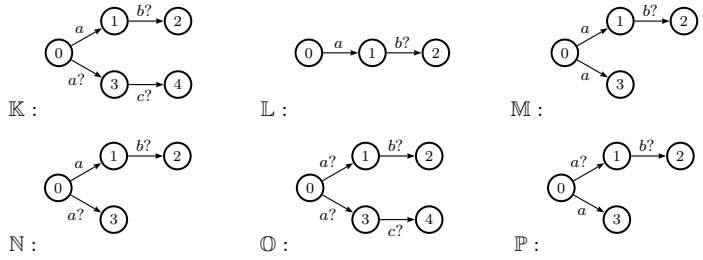


Fig. 11. Example MTSs for illustrating cover sets.

DEFINITION 6.2. (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 notion of a *Cover Set* – 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 M and N.

DEFINITION 6.3. (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) $M \preceq_{\mathrm{a}} Abs(P, \Delta^r_P(s, A) \setminus \zeta_{s,A})$
(3) $N \preceq_{\mathrm{a}} Abs(P, \Delta^r_P(s, A) \setminus \zeta_{s,A})$

The first rule of Cover Set 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. The second (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 11). $\zeta_{0,a} = \{0 \xrightarrow{a} 3\}$ 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, to compute a merge of models M and N, 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 models in Figure 11. 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 a model $\mathbb{N}$, and the result of $Abs(\mathbb{M}, \Delta^r_{\mathbb{M}}(0, a) \setminus \zeta'_{0,a})$ is a model $\mathbb{P}$. While $\mathbb{P}$ is a refinement of $\mathbb{N}$, $\mathbb{N}$ is *not* a refinement

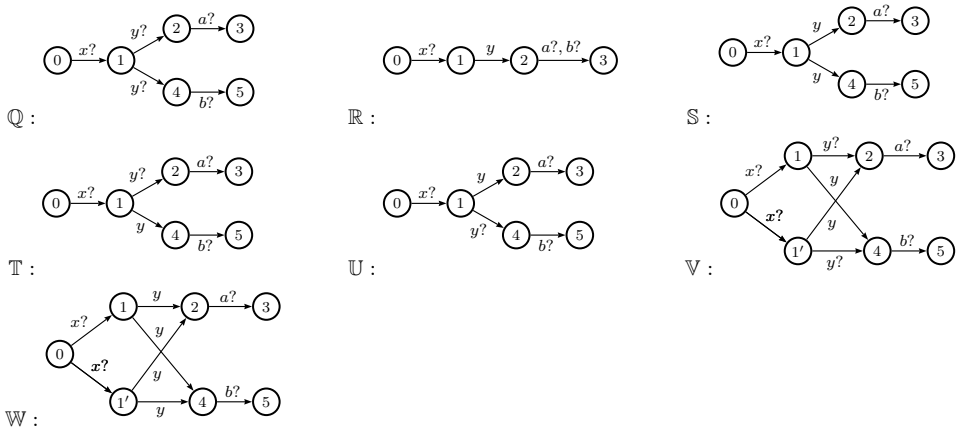


Fig. 12. Example MTSs for illustrating the need for cloning states.

of $\mathbb{P}$. Hence, out of the two choices of the cover set, 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 formalize this intuition below:

DEFINITION 6.4. (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 partial order, i.e., a common refinement may have two cover sets where neither refines the other. Consider the models in Figure 12. 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. 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 replicate, or 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 1 in order to abstract the model.

We formally define the clone operation below.

DEFINITION 6.5. (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) $\Delta^{r'} = \Delta^r \cup \{(s', \ell, t) | (s, \ell, t) \in \Delta^r\} \cup \{(t, \ell, s') | (t, \ell, s) \in \Delta^r\}$

PROPERTY 6.1. *The clone operation preserves implementations. In other words,*

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

ALGORITHM 6.1. 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

Fig. 13. The MERGE algorithm.

We are now ready to present the algorithm MERGE (see Algorithm 6.1 in Figure 13). 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 12, 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 tries to only build 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 compleity of the algorithm is discussed in Section 6.3. 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.

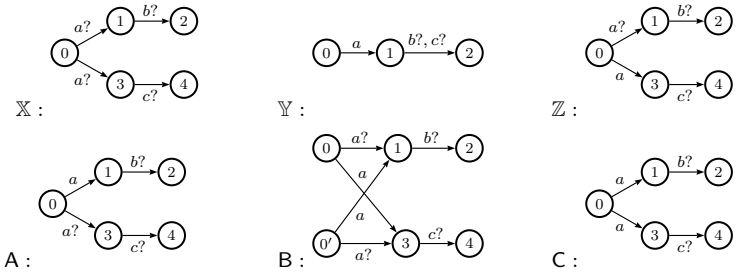


Fig. 14. Example of a merge where the LCR does not exist.

PROPERTY 6.2. *If there is a consistency relation between M and N, then the* MERGE *algorithm produces a common refinement of M and N but not necessarily a minimal one.*

As we discussed in Section 4, two consistent models do not always have a unique least common refinement – they may have non-equivalent minimal common refinements instead. In the remainder of this section, we show that the MERGE algorithm deals with these cases correctly and effectively when it clones states.

Consider the models in Figure 14 (model $\mathbb{X}$ of Figure 11 is repeated here for convenience). Model C is a common refinement of $\mathbb{X}$ and $\mathbb{Y}$ and has two incomparable cover sets: $\zeta_{0,a} = \{0 \xrightarrow{a} 1\}$ and $\zeta'_{0,a} = \{0 \xrightarrow{a} 3\}$. Interestingly, applying the abstraction operation of these incomparable cover sets on C yields two minimal common refinements of $\mathbb{X}$ and $\mathbb{Y}$: models $\mathbb{Z}$ and A. The result of the MERGE algorithm is model B (by cloning state 0 of C and applying a different cover set to each clone). While B is equivalent to $\mathbb{Z}$ because state $0'$ is unreachable, changing the initial state from 0 to $0'$ yields a model equivalent to the other MCR of $\mathbb{X}$ and $\mathbb{Y}$, namely, A.

Hence, the MERGE algorithm is able to encode the various ways in which the two models can be merged by computing an MTS with *a set* of potential initial states. Each one of these states defines an MTS which is an MCR of the models being merged (as in the above example) or which refines it. We formalize this correctness property below.

PROPERTY 6.3. *Let M and N be MTSs with a consistency relation between them. Let $P = (S_P, A_P, \Delta^r_P, \Delta^p_P, (m_0, n_0)^0)$ be the result of applying the* MERGE *algorithm to M and N, and let $(m_0, n_0)^j$ be the clones of the initial state of P. Then, for any $P_j = (S_P, A_P, \Delta^r_P, \Delta^p_P, (m_0, n_0)^j)$ where $(m_0, n_0)^j \in S_P$, there exists $Q \in \mathcal{MCR}(M, N)$ such that $Q \preceq P_j$.*

6.3 Limitations and Discussion

Unlike merge under strong semantics, described in [Fischbein and Uchitel, 2008], the MERGE algorithm in Section 6.2 is not complete: given two consistent MTSs with a unique LCR, it does not necessarily construct this LCR; further, given two consistent MTSs with multiple MCRs, it does not necessarily encode *all* of these in its resulting MTS. There are two reasons for this incompleteness: (1) reliance on the existence of a consistency relation which is not complete with respect to weak alphabet refinement (Theorem 5.2) and (2) the use of the insufficiently strong abstraction strategy based on cloning and cover sets.

In other words, the first reason for incompleteness is that in the case of weak alphabet refinement, the non-existence of a consistency relation does not imply inconsistency.

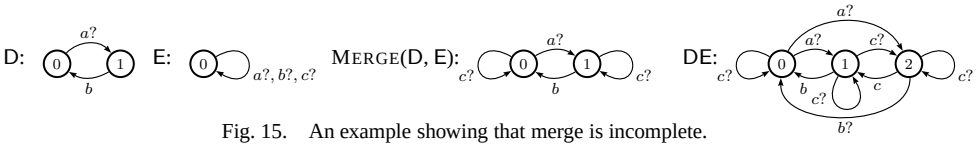


Fig. 15. An example showing that merge is incomplete.

.

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 refinement. The original merge operation for strong refinement, from [Larsen et al., 1996], was also incomplete in a similar way, requiring the existence of an independence relation. In addition, the algorithm presented in this section is an extension to that of [Fischbein and Uchitel, 2008] and hence is complete with respect to strong refinement.

The second cause for incompleteness is that the abstraction strategy based on cloning and cover sets is not sufficiently strong to guarantee that the LCR or MCRs will be reached through successive abstractions. For example, consider models D and E in Figure 15. Their common refinement, DE, is strictly more refined than MERGE(D, E).

In general, a complete merge algorithm is not possible. The reason for this is that two MTSs may have an infinite number of MCRs. Encoding such cases in a single MTS would result in a model with an infinite number of states.

While incomplete, the MERGE algorithm is better than the one presented in [Uchitel and Chechik, 2004]. For the cases handled by the algorithm in [Uchitel and Chechik, 2004], MERGE can compute more abstract common refinements. In addition, MERGE can compute common refinements (and possibly minimal common refinements) for a broader range of consistent MTSs.

Some sufficient rules for guaranteeing completeness have been explored in [Brunet, 2006]. However, they are either not intuitive enough to be practical to an engineer, or overly restrictive.

The time complexity of the merge algorithm strongly depends on the amount of non-determinism of the model produced by the application of the $+_{cr}$ operation. If this initial approximation of the merge is deterministic the time complexity is polynomial. However, it grows exponentially with the degree of non-determinism of the model. The degree of non-determinism is defined as follows: The degree of non-determinism of a model on a given state and label is equal to the number of outgoing transitions with that label minus one. The degree of non-determinism of an MTS is the sum of the degree of non-determinism for every state and label. Thus, our MERGE algorithm has the same complexity as the merge algorithm for strong refinement [Fischbein and Uchitel, 2008]. However, while both algorithms are exponential in time and polynomial in space, the degree of non-determinacy is higher for the cases of weak alphabet refinement, rendering the corresponding algorithm more expensive in practice.

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 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 10.3.

7. 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 it the same to merge two models and elaborate the result through refinement as it is to elaborate the models independently and then merge 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 merges can be made in order to guarantee particular algebraic properties. In Section 9, we apply these results to a case study.

7.1 Properties of Parallel Composition

We first study properties of the parallel composition operator proposed by Larsen in [Hüttel and Larsen, 1989]. 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.

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

$$\mathcal{I}[M||N] = \{I_M||I_N \mid I_M \in \mathcal{I}[M] \wedge I_N \in \mathcal{I}[N]\}) \quad (1)$$

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

However, this is not the case even under strong refinement. Consider the models in Figure 16. 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 N, this is not the case. Consider models in Figure 17. 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 [Hüttel and Larsen, 1989] produces a superset of the expected implementations (see Equation (1) above) independently of the choice of refinement (strong, weak, weak alphabet):

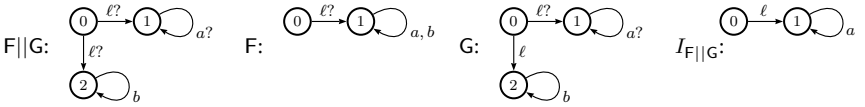


Fig. 16. Examples for Parallel Composition: Non-Deterministic Models.

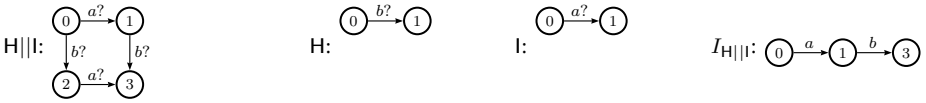


Fig. 17. Example for Parallel Composition: Deterministic Models with Different Alphabets.

THEOREM 7.1. (Implementations of the MTS Parallel Composition Operator) *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 correct and complete with respect to a semantic definition along the lines of the one proposed in Equation (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 7.2. (Parallel Composition Preserves Deterministic Implementations) *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 and weak alphabet refinement.

Even though the parallel composition operator admits more implementations than it should (Theorem 7.1), the following results provide guarantees of property preservation and give 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 [Milner, 1989]. A formal definition is presented below.

DEFINITION 7.1. (Simulation) *[Milner, 1989] Let LTSs P and Q such that $\alpha P = \alpha Q$. Q* simulates *M, written $P \sqsubseteq_{\mathrm{s}} Q$, iff (P, Q) is contained in some* simulation relation *$R \subseteq \wp \times \wp$, for which the following holds for all $\ell \in Act$ and for all $(P', Q') \in R$:*

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

THEOREM 7.3. (Parallel Composition Preserves Simulation) *Let M and N be MTSs and $I_{M||N}$ be an LTS. If $I_{M||N} \in \mathcal{I}_A[M||N]$, then*

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

Given that simulation relations preserve safety properties ([Abadi and Lamport, 1991]), 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.

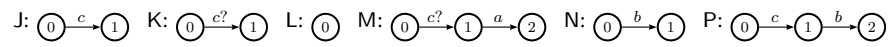


Fig. 18. Example MTSs for algebraic properties.

COROLLARY 7.1. (Parallel Composition Preserves True Safety Properties) *[Uchitel et al., 2009] Let M and N be MTSs and $\varphi \in$ FLTL. If φ is a safety property and $\|\varphi\|^M = \mathbf{t}$, then $\|\varphi\|^{M\|N} = \mathbf{t}$.*

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 7.2, 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.

7.2 Properties of LCRs

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

PROPERTY 7.1. *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 LCR 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 7.2. (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 $C \in \mathcal{MCR}(P, Q)$.*

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 and follows from Corollary 7.1 and the fact that parallel composition is monotonic under weak alphabet refinement.

PROPERTY 7.3. (Monotonicity 2) *If $M \preceq_a N$ and $\alpha P \subseteq \alpha M$, then $M\|P \preceq_a N\|P$.*

PROPERTY 7.4. *Let M, N, and P be MTSs such that $\alpha P \subseteq \alpha M \cap \alpha N$ and φ is a safety FLTL property. If $\|\varphi\|^{M\|P} = \mathbf{t}$ or $\|\varphi\|^{N\|P} = \mathbf{t}$, then $\|\varphi\|^{\mathcal{LCR}_{M,N}\|P} = \mathbf{t}$.*

7.3 Properties of MCRs

In this subsection, 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 7.2 that still holds as is, since an LCR always exists between a system and itself. The rest of the properties discussed in Section 7.2 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 7.5. (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* (see Definition 3.15) reachable through refinement is not affected by the merge order.

PROPERTY 7.6. (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 10.3.

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 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 18 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 7.7. (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 7.7 reduces to Property 7.3, 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.

Finally, Property 7.4 can also be extended to the context of multiple MCRs:

PROPERTY 7.8. *Let M, N, and P be MTSs such that $\alpha P \subseteq \alpha M \cap \alpha N$ and φ is a safety FLTL property. If $\|\varphi\|^{M\|P} = \mathbf{t}$ or $\|\varphi\|^{N\|P} = \mathbf{t}$, then $\forall A \in \mathcal{MCR}(M,N) \cdot \|\varphi\|^{A\|P} = \mathbf{t}$.*

In this subsection, we have shown that properties which hold for LCRs do not hold when consistent models have no unique MCR (Section 7.2). Intuitively, the existence of nonequivalent 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.

8. TOOL SUPPORT

We have developed a tool that supports construction and analysis of MTS models: the Modal Transition System Analyzer (MTSA) [D'Ippolito et al., 2008] (available at `http://sourceforge.net/projects/mtsa/files/mtsa/MTSA-R2/`). The basic mechanism for describing MTS models is using a text language based on the FSP process algebra [Magee and Kramer, 1999] 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, and various analyses such as animation, model checking of FLTL properties, consistency checking, as well as deadlock freedom and refinement checks.

The tool builds upon the Labelled Transition System Analyzer (LTSA) [Magee and Kramer, 1999], utilizing and extending its graphical user interface as well as specific analysis algorithms for LTSs. For instance, MTSA implements 3-valued FLTL model checking of MTSs (under inductive semantics) by reducing the problem to two classical FLTL model-checking runs on LTS models (see Theorem 3.3). Hence, MTSA builds on top of the model checking features of LTSA.

9. A CASE STUDY: THE MINE PUMP

We have applied the results described in this paper to a number of case studies including the Philips television product family [van Ommering et al., 2000] and use-case based specifications of information system.

The purpose of this section is to show, by means of the mine pump [Kramer et al., 1983] case study, how the results described earlier in this paper are exploited in an incremental behaviour model elaboration process. We do focus on multiple iterations nor on the

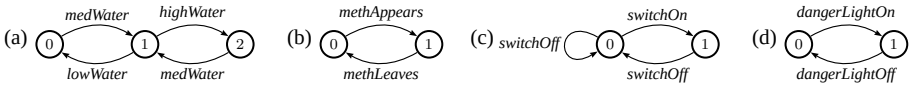


Fig. 19. The LTSs for (a) *WaterLevelSensor*, (b) *MethaneSensor*, (c) *Pump*, and (d) *DangerLight*

high-level languages that can be used to describe behaviour from which MTS models are synthesized as this is beyond the scope of this paper.

All analyses were performed automatically by means of the MTSA tool described above. In Section 9.1, we give a high level overview and introduce some components of the case study. In Section 9.2, we show how a behaviour model for the case study can be constructed by merging partial models of the intended system behaviour, and how tool-supported validation of the resulting model can prompt further elaboration. We construct the final model, which satisfies the expected requirements, through successive merge operations over partial models.

9.1 Informal Description

A pump controller is used to prevent the water in a mine sump 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. In addition, the pump is equipped with a danger light that is intended to reflect the presence of methane in the sump.

The mine pump system consists of five components: *WaterLevelSensor*, *MethaneSensor*, *DangerLight*, *PumpController*, and *Pump*. The complete system, *MinePumpSystem*, is the parallel composition of these components. *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, *Pump* and *DangerLight* model the physical pump and danger light, respectively, which can be switched on and off. The LTSs for these components are shown in Figure 19, where we assume that initially the water is low, the pump is off, no methane is present, and the danger light is off. *PumpController* describes the controller that monitors the water and methane levels, controls the pump in order to guarantee the properties of the mine pump system, and also maintains the status of the danger light according to the methane level.

The informal 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 models for the pump controller, which are intended to be merged to create a model of the entire system, namely, *MinePumpSystem*.

9.2 Model Construction and Elaboration

We begin by formalizing the requirements of the intended system behaviour given in Section 9.1 and providing the MTSs initially used to model partial operational views of this behaviour. We then show how the merging and analysis results presented earlier in this paper can support the elaboration of a final system model that satisfies the intended requirements.

9.2.1 *Initial Models and Properties.* Assume that requirements specification of the pump has been organized following the IEEE Recommended Practice for Software Re-

quirements Specifications Standard 830 [IEEE, 1994], which provides a template for structuring requirements based on the operation mode of the system-to-be. Consequently, requirements are grouped into those that are relevant when the mine pump is on and those in which the mine pump is off.

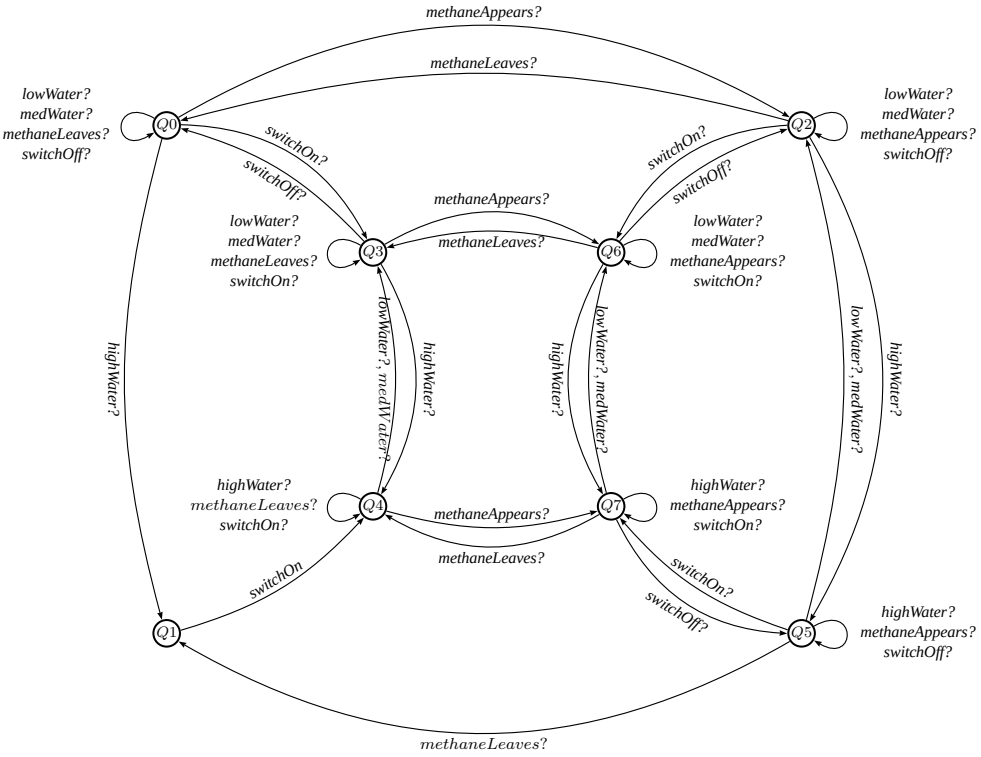
As with the ATM example presented in Section 2, the operational requirements for the mine pump controller could be given in a variety of specification languages. From some of these languages (e.g., MSCs, use cases or temporal logics), MTS models could be synthesized automatically [Uchitel et al., 2007; Uchitel et al., 2009]. Synthesis of MTSs is beyond the scope of this paper and consequently of this case study. Hence, we assume that two MTSs have been constructed manually or (semi-)automatically from the requirements corresponding to each mode.

Thus, we begin with two MTSs for the pump controller: one corresponding to the mode in which the pump is off and which specifies when the pump should be switched *on* (referred to as the "on policy", or *OnPolicy* – see Figure 20) and another – for the mode in which the pump is on and which specifies when the pump should be switched *off* (referred to as the "off policy", or *OffPolicy* – see Figure 21). While Figure 20 contains both a graphical depiction and a textual representation of *OnPolicy*, for larger models, such as *OffPolicy*, we use only a textual representation, as in Figure 21.

OnPolicy turns the pump on when there is high water and no methane present, and leaves open the possibility of turning the pump on when there is medium water. It keeps track of the state of the pump in addition to the level of methane and water in order to enable switching the pump on at the appropriate moment. The MTS depicted in Figure 20 for *OnPolicy* is easier understood by noting that the four states in the top half of the diagram correspond to the water being high, and those in the lower half correspond to the water not being high; the states on the left side model the case when there is no methane present, while the right side has states in which methane is present; and finally, the four states in the center correspond to the pump being on, while the outer four states are when the pump is off. Note state $Q1$ which requires *switchOn*; this state is central to enforcing the "on policy".

The *OffPolicy* turns the pump off when there is low water or methane present. To do this, it keeps track of the state of the pump and changes in the water and methane levels to force *switchOff* as soon as the water becomes low or methane appears. In addition, *OffPolicy* models a danger light with actions *dangerLightOn* and *dangerLightOff* (unobservable to *OnPolicy*), turning the light on when methane is present in the mine. The intention of the model is to guarantee that the danger light warns miners when methane is present; hence, the corresponding danger light action is the only action allowed upon changes in the level of methane.

Each of the models has been validated independently and found to correspond to the intended behaviour of the pump. The validation of these models may have included human-centric activities such as inspection and observation of animations and simulations, and automated verification techniques such as model checking each model individually against alternative specifications of the system. Specifically, we assume that the validation of the MTS models included properties that both models are expected to satisfy. These are Φ_1 and Φ_2, which express that the pump should only be turned on if it is already off, and should only be turned off if it is already on, respectively (see Table I). In addition, an expected property for the on policy is Φ_3, which states that when there is high water and no methane, the pump should be immediately turned on if it is not on already. An expected




```
OnPolicy = Q0,
  Q0  = ({lowWater?, medWater?,
       methaneLeaves?, switchOff?} -> Q0
    |highWater? -> Q1
    |methaneAppears? -> Q2
    |switchOn? -> Q3),
  Q1  = (switchOn -> Q4),
  Q2  = (methaneLeaves? -> Q0
    |{lowWater?, medWater?,
         methaneAppears?, switchOff?} ->
         Q2
    |highWater? -> Q5
    |switchOn? -> Q6),
  Q3  = (switchOff? -> Q0
    |{lowWater?, medWater?, methaneLeaves
         ?, switchOn?} -> Q3
    |highWater? -> Q4
    |methaneAppears? -> Q6),
  Q4  = ({lowWater?, medWater?} -> Q3
    |{highWater?, methaneLeaves?,
         switchOn?} -> Q4
    |methaneAppears? -> Q7),
  Q5  = (methaneLeaves? -> Q1
    |{lowWater?, medWater?} -> Q2
    |{highWater?, methaneAppears?,
         switchOff?} -> Q5
    |switchOn? -> Q7),
  Q6  = (switchOff? -> Q2
    |methaneLeaves? -> Q3
    |{lowWater?, medWater?,
         methaneAppears?, switchOn?} ->
         Q6
    |highWater? -> Q7),
  Q7  = (methaneLeaves? -> Q4
    |switchOff? -> Q5
    |{lowWater?, medWater?} -> Q6
    |{highWater?, methaneAppears?,
         switchOn?} -> Q7).
```

Fig. 20. The graphical and the textual representations of the MTS for *OnPolicy*.

property for the off policy is Φ_4, which states that if there is low water or methane present, the pump should be immediately turned off if it is not off already.

9.2.2 *Analysis.* Using MTSA [D'Ippolito et al., 2008], we verify that in *OnPolicy*, Φ_3 evaluates to *true* , but properties Φ_1, Φ_2 and Φ_4 evaluate to *maybe* (see Table II). We further determine that in *OffPolicy*, Φ_2 and Φ_4 evaluate to *true* , but properties Φ_1 and Φ_3 evaluate to *maybe* .

```
OffPolicy = Q0,
  Q0  = (lowWater? -> Q0
    |{highWater?, medWater?} -> Q1
    |methaneLeaves? -> Q15
    |methaneAppears? -> Q16),
  Q1  = (lowWater? -> Q0
    |{highWater?, medWater?} -> Q1
    |switchOn? -> Q2
    |methaneLeaves? -> Q13
    |methaneAppears? -> Q14),
  Q2  = (switchOff? -> Q1
    |{highWater?, medWater?, switchOn?}
        -> Q2
    |methaneLeaves? -> Q3
    |methaneAppears? -> Q4
    |lowWater? -> Q12),
  Q3  = (dangerLightOff -> Q2),
  Q4  = (dangerLightOn -> Q5),
  Q5  = (switchOff -> Q6),
  Q6  = ({highWater?, medWater?} -> Q6
    |methaneLeaves? -> Q7
    |methaneAppears? -> Q8
    |lowWater? -> Q9),
  Q7  = (dangerLightOff -> Q1),
  Q8  = (dangerLightOn -> Q6),
  Q9  = ({highWater?, medWater?} -> Q6
    |lowWater? -> Q9
    |methaneLeaves? -> Q10
    |methaneAppears? -> Q11),
  Q10 = (dangerLightOff -> Q0),
  Q11 = (dangerLightOn -> Q9),
  Q12 = (switchOff -> Q0),
  Q13 = (dangerLightOff -> Q1),
  Q14 = (dangerLightOn -> Q6),
  Q15 = (dangerLightOff -> Q0),
  Q16 = (dangerLightOn -> Q9).
```

Fig. 21. The MTS for *OffPolicy*.

$$
\begin{aligned}
\mathit{MonitoredActions} &\triangleq (\mathit{highWater} \vee \mathit{lowWater} \vee \mathit{medWater} \vee \mathit{methaneAppears} \vee \mathit{methaneLeaves}) \\
&\quad \text{(auxiliary definition: an action monitored by the pump controller has occurred)} \\
\Phi_1 &= \Box(\mathit{PumpOn} \Rightarrow \mathbf{X}(\neg \mathit{switchOn}\ \mathbf{W}\ \neg \mathit{PumpOn})) \\
&\quad \text{(the off policy: the pump is turned off when there is low water or methane present)} \\
\Phi_2 &= \Box(\neg \mathit{PumpOn} \Rightarrow \mathbf{X}(\neg \mathit{switchOff}\ \mathbf{W}\ \mathit{PumpOn})) \\
&\quad \text{(the pump can only be turned on if it is currently off)} \\
\Phi_3 &= \Box(\mathit{AtHighWater} \wedge \neg \mathit{MethanePresent} \Rightarrow \mathbf{X}(\neg \mathit{MonitoredActions}\ \mathbf{W}\ \mathit{PumpOn})) \\
&\quad \text{(the pump can only be turned off if it is currently on)} \\
\Phi_4 &= \Box(\mathit{AtLowWater} \vee \mathit{MethanePresent} \Rightarrow \mathbf{X}(\neg \mathit{MonitoredActions}\ \mathbf{W}\ \neg \mathit{PumpOn})) \\
&\quad \text{(the on policy: the pump is turned on when there is high water and no methane)}
\end{aligned}
$$

Table I. Desired properties of *MinePump*, expressed in FLTL.

Maybe values of Φ_4 on *OnPolicy* and of Φ_3 on *OffPolicy* are reasonable as neither model was produced with the initial goal of satisfying these properties. This gives us reason to believe that it is possible to construct an implementation that conforms to both policies and satisfies both properties. We construct such an implementation below by determining that the policies are consistent, merging them, and obtaining a model whose implementations satisfy the properties.

Interestingly, although all implementations of *OffPolicy* are guaranteed to satisfy Φ_2, only some are guaranteed to satisfy Φ_1! This is not necessarily a modelling error, rather, it can mean that assumptions are being made about the behaviour of the environment.

Specifically, the *Pump* model in Figure 19 does not allow the pump to be switched on when it is currently on; hence, it guarantees Φ_1, as verified by MTSA. This is sufficient to guarantee (using Corollary 7.1 and the fact that Φ is a safety property) that Φ_1 evaluates to true in *OffPolicy*$\|$*Environment*, where

$$\mathit{Environment} = \mathit{WaterLevelSensor}\|\mathit{MethaneSensor}\|\mathit{Pump}\|\mathit{DangerLight}.$$

While theoretical results guarantee that Φ_1 is true in *OffPolicy*$\|$*Environment*, we can check this using MTSA, obtaining the expected results (see Table II).

In summary, properties Φ_1 and Φ_2 evaluate to *maybe* in *OnPolicy*, and similarly to *OffPolicy*, the model of the *Pump* guarantees that any controller that conforms to *OnPolicy*

	Φ_1	Φ_2	Φ_3	Φ_4
OnPolicy	*maybe*	*maybe*	*true*	*maybe*
OffPolicy	*maybe*	*true*	*maybe*	*true*
Pump	*true*	*false*	-	-
Environment $\triangleq$ *Pump* $\|\|$ *MethaneSensor* $\|\|$ *WaterLevelSensor* $\|\|$ *DangerLight*	*true*	*false*	*false*	*false*
OnPolicy $\|\|$ *Environment*	*true*	*maybe*	*true*	*maybe*
OffPolicy $\|\|$ *Environment*	*true*	*true*	*maybe*	*true*
$PumpController_1 \triangleq$ *OnPolicy* $++$ *OffPolicy*	*maybe*	*true*	*true*	*true*
(*OnPolicy* $\|\|$ *Environment*) $++$ (*OffPolicy* $\|\|$ *Environment*)	*true*	*true*	*true*	*true*
$PumpController_1$ $\|\|$ *Environment*	*true*	*true*	*true*	*true*
$PumpController_2 \triangleq PumpController_1 ++$ *MethaneSensor* $++$ *WaterLevelSensor*	*maybe*	*true*	*true*	*true*
$PumpController_2$ $\|\|$ *Environment*	*true*	*true*	*true*	*true*

Table II. Property evaluation in different models.

satisfies Φ_1 when composed with the environment. However, neither *Environment* nor *OnPolicy* guarantee Φ_2, nor does their parallel composition.

9.2.3 *Merged Policies.* Since Φ_3 evaluates to *maybe* in *OffPolicy* and to *true* in *OnPolicy*, while Φ_4 evaluate to *true* in *OffPolicy* and *maybe* in *OffPolicy*, if the policies are consistent, that is, there are implementations that conform to both, then these implementations satisfy both properties. This is due to Corollary 4.1, since refinement preserves FLTL. In addition, since *OffPolicy* also satisfies Φ_2, LTSs that implement both policies satisfy this property as well.

Using MTSA, we can check that the two policies are consistent and build their merge, shown in Figure 22 (in MTSA terms, MERGE(M, N) is denoted M ++ N). MTSA can also be used to verify that properties Φ_2-Φ_4 are *true* (as guaranteed by Corollary 4.1) and Φ_1 is *maybe* in $PumpController_1$ = *OnPolicy* ++ *OffPolicy*.

However, what can be said about the truth of Φ_1? This property holds in *OffPolicy* $\|$ *Environment* and in *OnPolicy* $\|$ *Environment*. Hence, it also holds in their merge:

$$(\textit{OffPolicy}\|\textit{Environment}) \text{ ++ } (\textit{OnPolicy}\|\textit{Environment}). \tag{2}$$

Yet we are interested in building a model for the pump controller and composing this model with its environment. In other words, we wish to reason about

$$(\textit{OffPolicy} \text{ ++ } \textit{OnPolicy})\|\textit{Environment}. \tag{3}$$

By Property 7.8 (see Section 7.3), factoring out the parallel composition in (2) to obtain (3) is guaranteed to preserve true safety properties of (2), including Φ_1. We can use MTSA to verify that (*OffPolicy* ++ *OnPolicy*) $\|$ *Environment* is a refinement of (*OffPolicy* $\|$ *Environment*) ++ (*OnPolicy* $\|$ *Environment*), and that (*OffPolicy* ++ *OnPolicy*) $\|$ *Environment* satisfies properties Φ_1-Φ_4.

9.2.4 *Elaboration.* We can now construct the full model of the mine pump system by composing $PumpController_1$ in parallel with models of the water level sensor, the methane sensor and the pump:

$$MinePumpSystem_1 = PumpController_1\|\textit{Environment}.$$

The result, depicted in Figure 23, has many *maybe* transitions and admits deadlocking implementations. This can be checked using MTSA or by visual inspection, looking for reachable states without outgoing required transitions. This indicates a problem: The combined policies admit implementations of the pump controller which can deadlock if com-

```
PumpController1 = Q0,
  Q0  = (lowWater? -> Q0
      |medWater? -> Q7
      |methaneAppears? -> Q9
      |highWater? -> Q10
      |methaneLeaves? -> Q22),
  Q1  = (medWater? -> Q1
      |highWater? -> Q2
      |lowWater? -> Q3
      |methaneLeaves? -> Q13
      |methaneAppears? -> Q15),
  Q2  = (medWater? -> Q1
      |highWater? -> Q2
      |lowWater? -> Q3
      |methaneAppears? -> Q16
      |methaneLeaves? -> Q24),
  Q3  = (medWater? -> Q1
      |highWater? -> Q2
      |lowWater? -> Q3
      |methaneLeaves? -> Q14
      |methaneAppears? -> Q19),
  Q4  = (dangerLightOn -> Q1),
  Q5  = (dangerLightOn -> Q17),
  Q6  = (dangerLightOn -> Q18),
  Q7  = (lowWater? -> Q0
      |methaneAppears? -> Q4
      |medWater? -> Q7
      |highWater? -> Q10
      |switchOn? -> Q20
      |methaneLeaves? -> Q23),
  Q8  = (switchOff -> Q0),
  Q9  = (dangerLightOn -> Q3),
  Q10 = (switchOn -> Q21),
  Q11 = (dangerLightOff -> Q21),
  Q12 = (dangerLightOff -> Q20),
  Q13 = (dangerLightOff -> Q7),
  Q14 = (dangerLightOff -> Q0),
  Q15 = (dangerLightOn -> Q1),
  Q16 = (dangerLightOn -> Q2),
  Q17 = (switchOff -> Q2),
  Q18 = (switchOff -> Q1),
  Q19 = (dangerLightOn -> Q3),
  Q20 = (methaneAppears? -> Q6
      |switchOff? -> Q7
      |lowWater? -> Q8
      |methaneLeaves? -> Q12
      |{medWater?, switchOn?} -> Q20
      |highWater? -> Q21),
  Q21 = (methaneAppears? -> Q5
      |lowWater? -> Q8
      |methaneLeaves? -> Q11
      |medWater? -> Q20
      |{highWater?, switchOn?} -> Q21),
  Q22 = (dangerLightOff -> Q0),
  Q23 = (dangerLightOff -> Q7),
  Q24 = (dangerLightOff -> Q10).
```

Fig. 22. The MTS for $PumpController_1$.

```
System_It1 = Q0,
  Q0  = (methaneAppears? -> Q1
      |medWater? -> Q2),
  Q1  = (dangerLightOn -> Q3),
  Q2  = (lowWater? -> Q0
      |highWater? -> Q4
      |methaneAppears? -> Q5
      |switchOn? -> Q6),
  Q3  = (methaneLeaves? -> Q7
      |medWater? -> Q8),
  Q4  = (switchOn -> Q9),
  Q5  = (dangerLightOn -> Q8),
  Q6  = (switchOff? -> Q2
      |highWater? -> Q9
      |lowWater? -> Q10
      |methaneAppears? -> Q11),
  Q7  = (dangerLightOff -> Q0),
  Q8  = (lowWater? -> Q3
      |methaneLeaves? -> Q12
      |highWater? -> Q13),
  Q9  = (medWater? -> Q6
      |methaneAppears? -> Q14),
  Q10 = (switchOff -> Q0),
  Q11 = (dangerLightOn -> Q15),
  Q12 = (dangerLightOff -> Q2),
  Q13 = (medWater? -> Q8
      |methaneLeaves? -> Q16),
  Q14 = (dangerLightOn -> Q17),
  Q15 = (switchOff -> Q8),
  Q16 = (dangerLightOff -> Q4),
  Q17 = (switchOff -> Q13).
```

Fig. 23. The MTS for $MinePumpSystem_1$.

posed with the environment. Thus, we need to further refine the partial model of the pump controller, $PumpController_1$.

There is an implicit requirement not addressed by the partial model of the controller: it cannot block the environment inputs. The requirement can be satisfied by merging the controller model $PumpController_1$ with another MTS that captures this requirement, i.e., that has required transitions on the controller's inputs: water-level and methane events.

Merging $PumpController_1$ with the models for *WaterLevelSensor* and *MethaneSensor* achieves the desired result:

$$PumpController_2 = PumpController_1 \mathbin{++} WaterLevelSensor \mathbin{++} MethaneSensor$$

```
Controller2 = Q0,
  Q0  = (methaneAppears? -> Q4
      |medWater -> Q13),
  Q1  = (switchOff -> Q9),
  Q2  = (switchOn -> Q11),
  Q3  = (dangerLightOn -> Q1),
  Q4  = (dangerLightOn -> Q15),
  Q5  = (switchOff -> Q12),
  Q6  = (dangerLightOn -> Q9),
  Q7  = (dangerLightOn -> Q5),
  Q8  = (switchOff -> Q0),
  Q9  = (highWater? -> Q12
      |lowWater? -> Q15
      |methaneLeaves -> Q17),
  Q10 = (methaneAppears? -> Q3
      |lowWater? -> Q8
      |switchOn? -> Q10
      |highWater? -> Q11
      |switchOff -> Q13),
  Q11 = (methaneAppears? -> Q7
      |medWater -> Q10
      |switchOn? -> Q11),
  Q12 = (medWater -> Q9
      |methaneLeaves? -> Q14),
  Q13 = (lowWater -> Q0
      |highWater -> Q2
      |methaneAppears -> Q6
      |switchOn? -> Q10),
  Q14 = (dangerLightOff -> Q2),
  Q15 = (medWater -> Q9
      |methaneLeaves? -> Q16),
  Q16 = (dangerLightOff -> Q0),
  Q17 = (dangerLightOff -> Q13).
```

Fig. 24. The MTS for $PumpController_2$.

satisfies properties Φ_2-Φ_4. $PumpController_2$, depicted in Figure 24, also satisfies Φ_1 under parallel composition with *Environment*.

Finally, when composed in parallel with its entire environment, *Pump*, *MethaneSensor*, *WaterLevelSensor* and *DangerLight*, $PumpController_2$ results in an MTS which only has deadlock-free implementations. Thus, $PumpController_2$ is the desired model of the pump controller as all its implementations ensure the correct behaviour (non-deadlocking and conforming to Φ_1-Φ_4). The final mine pump system is

$$MinePumpSystem_2 = PumpController_2 \| Environment.$$

9.2.5 *Discussion.* The case study shows that by using MTSs, weak alphabet refinement and merge, we can support the elaboration of a system model from multiple partial models of the same system. This case study does not use all of the results discussed in this paper, as doing so would make the models too complex for this presentation. However, the case study does show the utility of various theoretical results described in previous sections and, in particular, results about combining merging with parallel composition, which is the standard operator for compositional construction of system models from complete (i.e., non-partial, two-valued) component models.

The distributivity property of merge and parallel composition is of particular relevance as it allows multiple partial models to be developed independently, with their own environmental assumption, while guaranteeing that the analysis remains valid under these assumptions as the partial models are merged.

Although the case study highlights how the theoretical results described in previous sections can be used to reason about the elaboration process, the tool support we have developed allows this reasoning to be done automatically. The models used in this case study, in the input format of MTSA, are available at `http://sourceforge.net/projects/mtsa/`.

10. CONCLUSIONS

In this section, we summarize the paper, compare our work with related approaches, and discuss directions for future research.

10.1 Summary

The motivation for the work presented in this paper comes from the need to support the elaboration of partial behaviour models. In particular, our research has been motivated by existing limitations for merging *different* partial behaviour models of the *same* system.

This paper studies merge for Modal Transition Systems which are a natural extension to Labelled Transition Systems that support partial behaviour descriptions. Although MTSs and merge have been studied extensively [Huth et al., 2001; Larsen and Thomsen, 1988; Larsen et al., 1996; Larsen et al., 1995; Fischbein and Uchitel, 2008; Uchitel et al., 2007; Uchitel et al., 2009], studies have included the strong assumption that alphabets of all models are the same. Hence, existing MTS semantics and merge, *strong* and *weak* [Larsen et al., 1996], do not allow for a more natural and realistic approach to modelling in which different viewpoints being merged have different scopes, and hence different alphabets, and in which the alphabet of the descriptions is extended as elaboration progresses.

In this paper, we present a study of Modal Transition Systems under a new semantics, called *weak alphabet semantics*, which supports alphabet elaboration together with behaviour elaboration. The paper makes a number of contributions including *(i)* a novel semantics for MTSs which preserves fluent linear temporal logic (FLTL) properties, *(ii)* theoretical and practical results regarding characterization of consistency which extend the current state of the art [Fischbein and Uchitel, 2008; Larsen et al., 1996], *(iii)* automated methods for constructing common refinements and merge, *(iv)* a study of the algebraic properties of merge and parallel composition and their relationship with refinement. These results give rise to a formal framework for partial model elaboration, based on merging and ensures that such a framework is adequately supported algorithmically.

10.2 Related Work

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

10.2.1 *Behaviour Modelling.* A significant body of work has been produced in the area of behaviour modelling, including research on process algebras (e.g., [Hoare, 1985]), notions of equivalence and refinement (e.g., [Milner, 1989]), and model checking (e.g., [Clarke et al., 1999]). The bulk of this work has used a two-valued semantics approach to behaviour modelling (e.g., using LTSs [Keller, 1976] 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 [Boem and Turner, 2004], and in processes that adopt use-case and scenario-based specifications (e.g., [CREWS, 1999; Uchitel et al., 2004]), or that are viewpoint-oriented [Hunter and Nuseibeh, 1998].

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 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.

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 [Chechik et al., 2003] and Mixed Transition Systems [Dams, 1996]. There is a trade-off between expressiveness, tractability and understandability and further studies, extending the results presented in this paper to these formalisms, are necessary.

10.2.2 *Merging.* Composition of behaviour models is not a novel idea [Milner, 1989; Hoare, 1985]; 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 [Cunningham and Finkelstein, 1986], aspects [Clarke et al., 2001], and scenario/use case composition (e.g., [Uchitel et al., 2003b; Krueger et al., 1999]). 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) [Larsen et al., 1995]. Their goal is to decompose a complete specification into several partial ones to enable compositional proofs. Although not studied in depth, the operator in [Larsen et al., 1995] is based on strong refinement. In particular, [Larsen et al., 1995; Larsen et al., 1996] use an incomplete notion of consistency and do not address the problem of multiple MCRs.

The subtleties of the existence of multiple MCRs under weak semantics were initially discussed in [Uchitel and Chechik, 2004] and then resolved for strong semantics in [Fischbein and Uchitel, 2008]: [Fischbein and Uchitel, 2008] presented a complete and correct merge algorithm for strong refinement together with a complete characterization of inconsistency under the same semantics. [Uchitel and Chechik, 2004] study merge and consistency for weak and weak alphabet semantics; however, the results presented in here are stronger: We characterize consistency under weak semantics and give a less restrictive precondition for consistency under weak alphabet semantics than the one given in [Uchitel and Chechik, 2004].

[Larsen and Xinxin, 1990] defines a conjunction operator for Disjunctive MTSs (DMTSs), similar to the one in [Larsen et al., 1995]. 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 [Larsen and Xinxin, 1990] is to characterize equation solving in process algebra. In particular, consistency is used to prove satisfiability of a given specification.

Hussain and Huth [Hussain and Huth, 2004] 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 paper 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. [Huth et al., 2002] use the mixed power-domain of Gunter [Gunter, 1992] 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 [Easterbrook and Chechik, 2001; Sabetzadeh and Easterbrook, 2003]. In [Easterbrook and Chechik, 2001] it is assumed that only states with the same label can be merged, and a similar consistency assumption is made in [Xing and Stroulia, 2005] in the context of UML differencing. On the other hand, in [Sabetzadeh and Easterbrook, 2003] 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 [Nejati and Chechik, 2005], 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 [Easterbrook and Chechik, 2001; Sabetzadeh and Easterbrook, 2003; Nejati and Chechik, 2005; Nejati et al., 2007] 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 [Harel et al., 2005] 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 [Krueger et al., 1999], program integration [Horwitz et al., 1989], and combining program summaries for software model-checking [Ball et al., 2004].

The notion of system composition through partial descriptions is at the core of approaches to feature interaction in telecommunication systems (e.g., [Calder and Magill, 2000; Nejati et al., 2008]). 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 paper (see [Nejati and Chechik, 2008] for details).

10.2.3 *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 [Larsen and Thomsen, 1988], and independently by [Dams, 1996]. Larsen and Thomsen introduced MTSs as a solution to the completeness limitation of LTSs, and proved that Hennessy-Milner logic [Hennessy and Milner, 1985] characterizes strong refinement. Dams' Mixed Transition Systems [Dams, 1996; Dams et al., 1997], 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 [Dams, 1996]. Bruns and Godefroid introduced partial Kripke Structures (PKs) [Bruns and Godefroid, 1999], 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.

[Huth et al., 2002] 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.

When a property evaluates to *maybe* in an abstract model, the model must be further refined (where refinement corresponds to splitting abstract states). [Shoham and Grumberg, 2004] show that even standard methods of refining abstract models (e.g. [Godefroid et al., 2001]) 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 [Godefroid and Jagadeesan, 2003], and in addition, 4-valued Kripke structure, Mixed Transition Systems and Generalized Kripke MTSs have the same expressive power as well [Gurfinkel et al., 2006; Wei et al., 2009].

10.3 Future Work

Our long-term goal is to provide a sound engineering approach to the development of software systems via automated support for constructing partial behaviour models from scenario-based specifications, merging and elaborating these partial behaviour models, as well as enabling users to choose the desired merge from the set of possible minimal common refinements. In particular, we plan to develop synthesis algorithms for constructing behaviour models from heterogeneous specifications (e.g. scenarios, properties, state-machines) and integrate this into our approach to merging partial behaviour models. First steps towards this goal are reported in [Uchitel et al., 2009].

We intend to continue experimentation by conducting larger case studies in order to further explore the limitations and the opportunities of the presented framework. One of the aspects to be addressed in the near future relates the practical difficulties introduced by merging models with no least common refinement. We aim to develop 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. It would also prevent the propagation of any incompleteness introduced by merging models. The fact that MTS are not closed under merge, i.e., that multiple MCRs may exist, also prompts the question of whether other partial behaviour modelling formalisms could be developed to better support incremental behaviour model elaboration.

Acknowledgements

The authors would like to thank Shoham Ben-David and Ivo Krka for carefully reading previous drafts of this manuscript. We are also grateful to Arie Gurfinkel for many fruitful discussions (specifically, about semantics of FLTL) and to anonymous TOSEM reviewers for their excellent suggestions.

REFERENCES

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

Ball, T., Levin, V., and Xie, F. (2004). "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.

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

Brunet, G. (2006). "A Characterization of Merging Partial Behavioural Models". Master's thesis, University of Toronto, Department of Computer Science.

Bruns, G. and Godefroid, P. (1999). "Model Checking Partial State Spaces with 3-Valued Temporal Logics". In *Proceedings of Proceedings of 11th International Conference on Computer-Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 274–287. Springer.

Bruns, G. and Godefroid, P. (2000). "Generalized Model Checking: Reasoning about Partial State Spaces". In *Proceedings of 11th International Conference on Concurrency Theory (CONCUR'00)*, volume 1877 of *LNCS*, pages 168–182. Springer.

Calder, M. and Magill, E. H., editors (2000). *Feature Interactions in Telecommunications and Software Systems VI, May 17-19, 2000, Glasgow, Scotland, UK*. IOS Press.

Chechik, M., Devereux, B., Easterbrook, S., and Gurfinkel, A. (2003). "Multi-Valued Symbolic Model-Checking". *ACM Transactions on Software Engineering and Methodology*, 12(4):1–38.

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

Clarke, E., Grumberg, O., and Peled, D. (1999). *Model Checking*. MIT Press.

Clarke, E. and Wing, J. (1996). "Formal Methods: State of the Art and Future Directions". *ACM Computing Surveys*, 28(4):626–643.

CREWS (1999). Cooperative Requirements Engineering With Scenarios. `http://Sunsite.Informatik.RWTH-Aachen.DE/CREWS`.

Cunningham, J. and Finkelstein, A. (1986). "Formal Requirements Specification: the FOREST Project". In *Proceedings of 3rd International Workshop on Software Specification and Design*, pages 186–192. IEEE CS Press.

Dams, D. (1996). *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, The Netherlands.

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

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

D'Ippolito, N., Fishbein, D., Chechik, M., and Uchitel, S. (2008). "MTSA: The Modal Transition System Analyzer". In *Proceedings of International Conference on Automated Software Engineering (ASE'08)*, pages 475–476.

Dupont, P., Lambeau, B., Damas, C., and van Lamsweerde, A. (2008). "The QSM Algorithm and its Application to Software Behavior Model Induction". *Journal of Applied Artificial Intelligence*, 22(1&2):77–115.

Dwyer, M. B., Avrunin, G. S., and Corbett, J. C. (1998). "Property Specification Patterns for Finite-state Verification". In *Proceedings of 2nd Workshop on Formal Methods in Software Practice*.

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

Fischbein, D. and Uchitel, S. (2008). "On Correct and Complete Merging of Partial Behaviour Models". In *Proceedings of SIGSOFT Conference on Foundations of Software Engineering (FSE'08)*, pages 297–307.

Fischbein, D., Uchitel, S., and Braberman, V. A. (2006). "A Foundation for Behavioural Conformance in Software Product Line Architectures". In *Proceedings of ISSTA'06 Workshop on Role of Software Architecture for Testing and Analysis (ROSATEA'06)*, pages 39–48.

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

Giannakopoulou, D. and Magee, J. (2003). "Fluent Model Checking for Event-Based Systems". In *Proceedings of the 9th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'03)*, pages 257–266. ACM Press.

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

Godefroid, P. and Jagadeesan, R. (2003). "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.

Godefroid, P. and Pitterman, N. (2009). "LTL Generalized Model Checking Revisited". In *Proceedings of the 10th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'09)*, volume 5403 of *LNCS*, pages 89–104.

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

Gurfinkel, A. and Chechik, M. (2005). "How Thorough is Thorough Enough". In *Proceedings of 13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'05)*, volume 3725 of *LNCS*, pages 65–80, Saarbrücken, Germany. Springer.

Gurfinkel, A., Wei, O., and Chechik, M. (2006). "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. Springer.

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

Hennessy, M. and Milner, R. (1985). "Algebraic Laws for Nondeterminism and Concurrency". *Journal of ACM*, 32(1):137–161.

Hoare, C. (1985). *Communicating Sequential Processes*. Prentice-Hall, New York.

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

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

Hussain, A. and Huth, M. (2004). "On Model Checking Multiple Hybrid Views". In *Proceedings of 1st International Symposium on Leveraging Applications of Formal Methods*, pages 235–242.

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

Huth, M., Jagadeesan, R., and Schmidt, D. A. (2001). "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.

Hüttel, H. and Larsen, K. G. (1989). "The Use of Static Constructs in A Modal Process Logic". In *Proceedings of Symposium on Logical Foundations of Computer Science (Logic at Botik'89)*, volume 363 of *LNCS*, pages 163–180.

IEEE (1994). "IEEE Recommended Practice for Software Requirements Specifications Standard 830". Technical Standard 830, Wallace S. Read (Chair).

ITU-T (1993). "ITU-T Recommendation Z.120: Message Sequence Chart (MSC)". *ITU-T*.

Jacobson, I. (2004). *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

Kazhamiakin, R., Pistore, M., and Roveri, M. (2004). "Formal Verification of Requirements using SPIN: A Case Study on Web Services". In *Proceedings of International Conference on Software Engineering and Formal Methods (SEFM'04)*, pages 406–415.

Keller, R. (1976). "Formal Verification of Parallel Programs". *Communications of the ACM*, 19(7):371–384.

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

Kramer, J., Magee, J., and Sloman, M. (1983). "CONIC: an Integrated Approach to Distributed Computer Control Systems". *IEE Proceedings*, 130(1):1–10.

Krueger, I., Grosu, R., Scholz, P., and Broy, M. (1999). "From MSCs to Statecharts". In Rammig, F. J., editor, *Distributed and Parallel Embedded Systems*. Kluwer Academic Publishers.

Larsen, K., Steffen, B., and Weise, C. (1996). "The Methodology of Modal Constraints". In *Formal Systems Specification*, volume 1169 of *LNCS*, pages 405–435. Springer.

Larsen, K. and Thomsen, B. (1988). "A Modal Process Logic". In *Proceedings of 3rd Annual Symposium on Logic in Computer Science (LICS'88)*, pages 203–210. IEEE Computer Society Press.

Larsen, K. and Xinxin, L. (1990). "Equation Solving Using Modal Transition Systems". In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science (LICS'90)*, pages 108–117. IEEE Computer Society Press.

Larsen, K. G., Steffen, B., and Weise, C. (1995). "A Constraint Oriented Proof Methodology based on Modal Transition Systems". In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'95)*, volume 1019 of *LNCS*, pages 13–28. Springer.

Letier, E., Kramer, J., Magee, J., and Uchitel, S. (2008). Deriving event-based transition systems from goal-oriented requirements models. *Autom. Softw. Eng.*, 15(2):175–206.

Magee, J. and Kramer, J. (1999). "*Concurrency - State Models and Java Programs*". John Wiley.

Milner, R. (1989). *Communication and Concurrency*. Prentice-Hall, New York.

Nejati, S. and Chechik, M. (2005). "Let's Agree to Disagree". In *Proceedings of 20th IEEE International Conference on Automated Software Engineering (ASE'05)*, pages 287 – 290. IEEE Computer Society.

Nejati, S. and Chechik, M. (2008). "Behavioural Model Fusion: Experiences from Two Telecommunication Case Studies". In *Proceedings of ICSE'08 Workshop on Modeling in Software Engineering (MiSE'08)*.

Nejati, S., Chechik, M., Sabetzadeh, M., Uchitel, S., and Zave, P. (2008). "Towards Compositional Synthesis of Evolving Systems". In *Proceedings of SIGSOFT Conference on Foundations of Software Engineering (FSE'08)*, pages 285–296.

Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., and Zave, P. (2007). "Matching and Merging of Statecharts Specifications". In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pages 54–64.

Pnueli, A. (1977). "The Temporal Logic of Programs". In *Proceedings of 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57.

Sabetzadeh, M. and Easterbrook, S. (2003). "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.

Shoham, S. and Grumberg, O. (2004). "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.

Sibay, G., Uchitel, S., and Braberman, V. (2008). "Existential Live Sequence Charts Revisited". In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 41–50. ACM.

Uchitel, S., Broy, M., Krueger, I. H., and Whittle, J. (2005). Guest editorial: Special section on interaction and state-based modeling. *IEEE Transactions on Software Engineering*, 31(12):997–998.

Uchitel, S., Brunet, G., and Chechik, M. (2007). "Behaviour Model Synthesis from Properties and Scenarios". In *Proceedings of International Conference on Software Engineering (ICSE'07)*, pages 34–43.

Uchitel, S., Brunet, G., and Chechik, M. (2009). "Synthesis of Partial Behaviour Models from Properties and Scenarios". *IEEE Transactions on Software Engineering*, 3(35):384–406.

Uchitel, S. and Chechik, M. (2004). "Merging Partial Behavioural Models". In *Proceedings of 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 43–52.

Uchitel, S., Kramer, J., and Magee, J. (2003a). "Behaviour Model Elaboration using Partial Labelled Transition Systems". In *Proceedings of the 9th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'03)*, pages 19–27.

Uchitel, S., Kramer, J., and Magee, J. (2003b). "Synthesis of Behavioural Models from Scenarios". *IEEE Transactions on Software Engineering*, 29(2):99–115.

Uchitel, S., Kramer, J., and Magee, J. (2004). "Incremental Elaboration of Scenario-Based Specifications and Behaviour Models using Implied Scenarios". *ACM Transactions on Software Engineering and Methodology*, 13(1):37–85.

van Lamsweerde, A. (2004). Goal-oriented requirements enginering: A roundtrip from research to practice. In *RE*, pages 4–7. IEEE Computer Society.

van Lamsweerde, A. and Letier, E. (2000). "Handling Obstacles in Goal-Oriented Requirements Engineering". *IEEE Transactions on Software Engineering*, 26(10):978–1005.

van Ommering, R., van der Linden, F., Kramer, J., and Magee, J. (2000). "The Koala Component Model for Consumer Electronics Software". *IEEE Computer*, 33(3):78–85.

Wei, O., Gurfinkel, A., and Chechik, M. (2009). "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.

Xing, Z. and Stroulia, E. (2005). "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.