Synthesis of Partial Behaviour Models from Properties and Scenarios

Sebastian Uchitel, Greg Brunet, and Marsha Chechik

Abstract— **Synthesis of behaviour models from software development artifacts such as scenario-based descriptions or requirements specifications helps reduce the effort of model construction. However, the models favoured by existing synthesis approaches are not sufficiently expressive to describe both universal constraints provided by requirements and existential statements provided by scenarios. In this paper, we propose a novel synthesis technique that constructs behaviour models in the form of Modal Transition Systems (MTS) from a combination of safety properties and scenarios. MTSs distinguish required, possible and proscribed behaviour, and their elaboration not only guarantees the preservation of the properties and scenarios used for synthesis but also supports further elicitation of new requirements.**

Index Terms— **Modal Transition Systems, Merge, Synthesis, Partial Behaviour Models.**

I. Introduction

Pre-development and pre-deployment reasoning about system behaviour supports identifying flaws early in the development process, greatly aiding the requirements and design processes. Labelled Transition Systems (LTSs) and other event-based behaviour models are convenient formalisms for modelling and reasoning about system behaviour. These models provide a basis for a wide range of automated analysis techniques, such as model-checking and simulation.

One of the serious limitations of behaviour modelling and analysis is the complexity of *building* the models in the first place. This process is somewhat simplified by compositionality: behaviour models describe a system as a set of interacting components where each component is modelled as a state machine, and interactions between components occur through shared events. Hence, behaviour models for complex systems can be constructed by describing simpler subsystems and composing them into more complex ones.

Despite compositionality, behaviour model construction remains a difficult, labour-intensive task that requires considerable expertise. To address this, a wide range of techniques for supporting (semi-)automated synthesis of behaviour models have been investigated. In particular, synthesis from declarative requirements specifications (e.g., [30], [37], [50], [43], [25]) or from scenarios and use cases (e.g., [29], [49], [27], [6]) has been studied extensively.

The first author is in Department of Computing, Imperial College, 180 Queen's Gate, London, SW7 2RH, UK, s.uchitel@doc.ic.ac.uk, and Department of Computer Science, FCEN, University of Buenos Aires, Argentina, suchitel@dc.uba.ar. The second author participated in early work leading to this paper while he was a graduate student at the University of Toronto and a research associate at Imperial College. The third author is in Department of Computer Science, University of Toronto, Toronto, Canada, chechik@cs.toronto.edu.

Synthesis from declarative specifications such as goal models describing the requirements of a system delivers executable models early in the requirements process, enabling a wide range of validation analyses such as animations, simulations, and scenario-based techniques.

Properties are statements that prune the space of acceptable behaviours of the system being built: The fact that a trace satisfies a property does not mean that the system is required to provide that trace; the trace could be violating another property, possibly one yet to be elicited. Consequently, a behaviour model synthesized from properties should characterize all possible behaviours that do not violate the properties. Such a model provides an *upper bound* on all the behaviours that the system will actually provide, once implemented. In other words, the final system cannot provide *more* behaviour than that described by the synthesized model, or, more formally, the synthesized model must be able to simulate the behaviour of the final system. Validation of behaviour models synthesized from properties can prompt the elicitation of more properties, which in turn will further approximate from above the intended behaviour of the system to be. In other words, as new properties are elicited, the resulting synthesized model will be able to do *less*, describing behaviour that is closer to that of the system to be.

Synthesis from scenario-based specifications such as Message Sequence Charts (MSCs) [24] has a number of advantages that complement those of property synthesis. Scenarios describe how system components, the environment and users interact in order to provide system-level functionality. Their simplicity and intuitive graphical representation facilitate stakeholder involvement and make them popular for requirements elicitation.

In their simplest, and widely used form, scenarios are existential statements: they provide examples of the intended system behaviour; in other words, sequences of interactions that the system is expected to exhibit (and do not, as opposed to properties, provide universal requirements for all system behaviours). By synthesizing behaviour models from scenarios, it is possible to support early analysis, validation, and incremental elaboration of behaviour models.

Scenarios are inherently partial descriptions, it is not normally assumed that the behaviour they describe amounts to the complete system behaviour. Consequently, a behaviour model synthesized from scenarios should provide a *lower bound* from which to identify the behaviours that the system will provide but that have not been explicitly captured by the scenarios. In other words, the synthesized model describes *less* behaviour than what the final system shall provide, or formally, the system shall be able to simulate the behaviour described by the scenarios. As new scenarios are identified, the resulting model will approximate the behaviour of the final system *from below*, meaning that the synthesized model will have *more* behaviour than the previous one (i.e., it will be able to simulate the previous one), but will still be a lower bound for the final system.

In this paper, we argue that classical state machine models (the target for existing synthesis approaches) such as LTSs are insufficiently expressive to support synthesis from *both* properties and scenarios as they cannot model simultaneously lower and upper bounds to intended system behaviour. We extend existing LTS synthesis algorithms to produce Modal Transition Systems (MTSs) [35] and demonstrate that elaboration from MTSs not only preserves the original properties and scenarios, but also supports elicitation of new properties and scenarios. In addition to the approach itself, specific contributions of the paper are:

1) a technique for automatically generating MTSs from safety properties expressed in Fluent Linear Temporal Logic (FLTL);
2) a technique for extending LTS synthesis from scenario approaches to support construction of MTS models;
3) a demonstration that *merging* of synthesized MTSs [48], preserves properties and scenarios and hence can be used as the basis for a combined synthesis procedure of behaviour models from all these artifacts. Given that merge characterizes, through refinement, all MTSs (and LTSs) that preserve the scenarios and properties, merge produces the starting point for further elaboration of system behaviour.
4) a report on a case study that demonstrates that analysis of synthesized MTSs can help find new meaningful properties and scenarios to be used to further the behaviour model elaboration process.

The rest of the paper is organized as follows. We begin with a motivating example in Section II, followed by the necessary background in Section III. In Section IV, 3-valued FLTL is presented. Sections V and VI present algorithms for synthesizing MTSs from safety properties and scenarios, respectively. In Section VII, we use the merge operator to put such partial behaviour descriptions together. Automated support for model synthesis techniques described in this paper is implemented in a tool called MTSA, which we describe in Section VIII. In Section IX, we apply results of this paper, illustrating construction of a partial model and its elaboration, identifying new scenarios and properties. We discuss our work and compare it to related approaches in Section X, and conclude in Section XI.

II. Motivating Example

In this section, we provide a motivating example, explaining the concepts of scenarios, properties, LTSs and synthesis informally.

Consider a simple web-based email system. Figure 1 provides some examples of the intended system behaviour using a standard message sequence chart notation [24]. The scenario sc describes a repetition (the outer *rep* box) of a (non-deterministic) choice (the inner *alt* box) between two sequences of actions: (1) a User requests authentication from

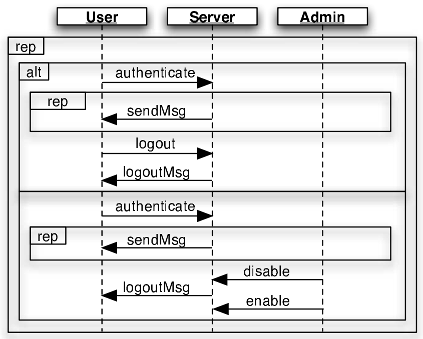


Fig. 1. Webmail scenario specification sc.

$$Registered = \langle enable, disable\rangle \text{ initially } TRUE$$
$$LoggedIn = \langle authenticate, \{logout, disable\}\rangle \text{ initially } FALSE$$

$$\begin{array}{rrcl} \text{(Legal access)} & p_1 & = & \Box(LoggedIn \Rightarrow Registered) \\ \text{(Private access)} & p_2 & = & \Box(sendMsg \Rightarrow LoggedIn) \\ \text{(Logouts are ack'd)} & p_3 & = & \Box(logout \Rightarrow \mathbf{X}\ logoutMsg) \end{array}$$

Fig. 2. Webmail system properties.

the Server which then sends a number of messages; after that, the User logs out and receives a logout message. (2) an Admin disables the User during user activities, effectively expelling the latter from the system. An example of a sequence of events required by sc is

$$sc_1 = authenticate, sendMsg, disable, logoutMsg \ldots$$

The Webmail system is required to enforce legal and private access to the emails it stores. These requirements are formalized in FLTL [13] in Figure 2 as properties p_1 and p_2. Legal access requires the User be *Registered* if she is to be *LoggedIn*. Private access requires that the User be *LoggedIn* if she is to receive e-mail (*sendMsg*) from the Server. *Registered* and *LoggedIn* are fluents that change value according to the occurrence of events. A User is *Registered* once he has been *enabled* and not yet *disabled*. A User is *LoggedIn* once he has been *authenticate*d and not yet done a *logout* nor has been *disabled*. An additional requirement, p_3, specifies that users should be sent an acknowledgment on logout. Formalization of p_3 states that if *logout* occurs, then the next ($\mathbf{X}$) event to occur is *logoutMsg*.

We now consider synthesis of LTS models from the scenarios and properties of the Webmail system.

Property $P = p_1 \wedge p_2 \wedge p_3$ can be used to synthesize, via an adaptation of the method in [13], an LTS model $L(P)$ shown in Figure 3. This model describes *all* possible behaviours over the events

$$Act_{web} = \{enable, disable, authenticate, logout, sendMsg, logoutMsg\}$$

that do not violate P. If P represents a subset of the actual system requirements, then the model $L(P)$ can be thought of as providing an *upper bound* on the actual intended behaviour of the system, and the elaboration process is aimed at removing behaviour from $L(P)$.

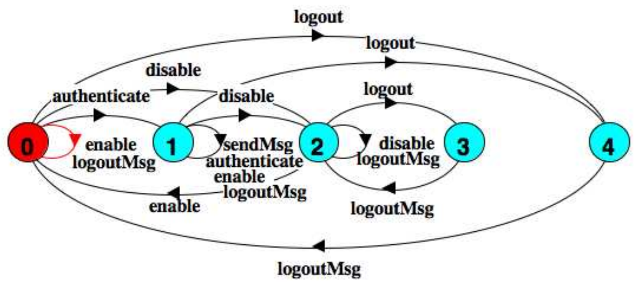


Fig. 3. LTS synthesized from property $P = p_1 \wedge p_2 \wedge p_3$.

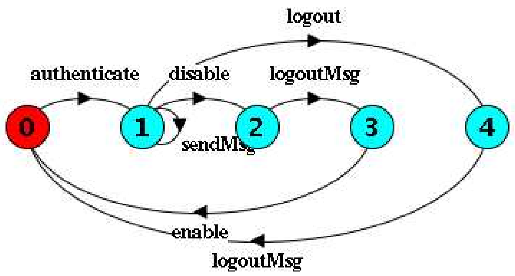


Fig. 4. LTS synthesized from scenario sc.

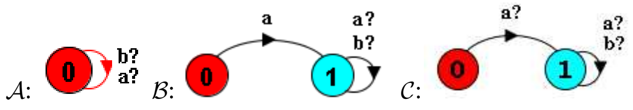


Fig. 5. Example MTSs where $\mathcal{A} + \mathcal{B} = \mathcal{B}$ and $\mathcal{A} \| \mathcal{B} = \mathcal{C}$.

The problem with $L(P)$, and with synthesis of LTSs in general, is that the model blurs the distinction between behaviours that *may* occur as they will not violate P, and behaviours that *must* occur in order to avoid a violation of P. For instance, it does not convey that removing a self-loop on *logoutMsg* from state 0 does not violate P, whereas removing a transition on the same event between states 4 and 0 does. Consequently, elaboration by arbitrary removal of behaviour can be incorrect. Furthermore, the problem of lack of distinction between *required* and *possible* behaviour is aggravated when the scenario description in Figure 1 is considered as well. From sc we know that removing the transition on *authenticate* from 0 to 1 would be incorrect as it would impede sc_1 from occurring; however, $L(P)$ does not, and cannot be modified to reflect this. In summary, the problem is that by interpreting the LTS $L(P)$ as an upper bound to the actual intended system behaviour, the distinction of what behaviour is required is lost. Such is the case of the transition on *authenticate* between states 0 and 4 and the self-loop on *logoutMsg* in state 1.

Synthesis of LTS models from scenarios presents the dual problem. A scenario description specifies only some of the required traces of the system. For example, Figure 1 says nothing about the possibility of the Admin disabling a User while the latter is not logged into the system:

$$sc_2 = \mathit{disable}, \mathit{enable}, \mathit{disable}, \mathit{enable}, \ldots$$

or the possibility of the User receiving messages after she has been disabled:

$$sc_3 = \mathit{authenticate}, \mathit{disable}, \mathit{sendMsg}, \mathit{logoutMsg}, \ldots$$

Such behaviours, although not explicitly required, could still be possible.

Synthesis from scenarios aims to build models that precisely capture the traces described by the scenarios. For example, Figure 4 depicts the LTS $L(sc)$ synthesized from the Webmail scenario sc using the algorithm described in [49]. Since scenario descriptions are partial, it is expected that the final LTS for the Webmail system will include all traces of $L(sc)$ as well as others. Hence, $L(sc)$ can be thought of as providing a *lower bound* of the intended system behaviour. The problem is, however, that not all LTS models that include the traces of $L(sc)$ are reasonable. For instance, the final LTS may include the trace sc_2 but not sc_3 since the latter violates the requirement ($p_1 \wedge p_2 \wedge p_3$) of the system. LTSs synthesized from scenarios cannot capture such restrictions.

To summarize, a major limitation of synthesis approaches is that the models being synthesized are assumed to be complete descriptions of the system behaviour with respect to a fixed alphabet of actions. Given the partial nature of the synthesis inputs (i.e., properties and scenarios), this forces the models to be interpreted as either lower or upper bounds of the intended system behaviour. Traditional behaviour models such as LTSs cannot capture in one model the middle ground, i.e., the behaviour that does not violate safety properties yet has not been required by scenarios, and this hinders validation, analysis and elaboration of behaviour models.

In this paper, we show how the limitations of existing synthesis techniques can be overcome by synthesizing more expressive behaviour models, namely, Modal Transition Systems (MTSs) [35], which are capable of distinguishing possible from required behaviour. We also show how analysis of possible but not required behaviour modeled in an MTS can support behaviour model elaboration.

III. Background

In this section, we review the notion of transition systems and operations over them, fix the notation and review merging of MTSs. For the ease of presentation, we assume that all transition systems have the same alphabet and do not use non-observable (τ) actions. For a treatment of models with different alphabets, please refer to [3].

A. Transition Systems

Definition 1: (Labelled Transition System) Let *States* be a universal set of states, and *Act* be a universal set of observable action labels. An *LTS* is a tuple $L = (S, \mathit{Act}, \Delta, s_0)$, where $S \subseteq \mathit{States}$ is a finite set of states, *Act* is the set of labels, $\Delta \subseteq (S \times \mathit{Act} \times S)$ is a transition relation, and $s_0 \in S$ is the initial state. We write $L \xrightarrow{\ell} L'$ if $(s_0, \ell, s_0') \in \Delta$ and $L' = (S, \mathit{Act}, \Delta, s_0')$.

An LTS models the interactions of a (sub-)system with its environment. An example LTS is shown in Figure 3. We use a convention that the initial state is labeled as 0. Otherwise, the numbers on states are for reference only and have no semantics. Transitions labelled with several actions is a shorthand for representing an individual transition on each action.

Definition 2: (Simulation) Let $\wp_L$ be the universe of all LTSs. An LTS L_0 *simulates* an LTS K_0, written $K_0 \leq L_0$, if there exists some simulation relation $R \subseteq \wp_L \times \wp_L$ such that $(K_0, L_0) \in R$ and $\forall \ell \in Act$, $\forall (K, L) \in R$,

$$(K \xrightarrow{\ell} K') \implies (\exists L' \cdot L \xrightarrow{\ell} L' \ \wedge \ (K', L') \in R)$$

Modal Transition Systems (MTSs) [35], which allow for explicit modelling of what is *not* known, extend LTSs with an additional set of transitions that model interactions with the environment that the system cannot be guaranteed to provide, and equally cannot be guaranteed to prohibit.

Definition 3: (Modal Transition System) An *MTS M* is a structure $(S, Act, \Delta^r, \Delta^p, s_0)$, where $\Delta^r \subseteq \Delta^p$, (S, Act, Δ^r, s_0) is an LTS representing *required* transitions of the system and (S, Act, Δ^p, s_0) is an LTS representing *possible* (but not necessarily required) transitions.

Every LTS (S, Act, Δ, s_0) can be embedded into an MTS $(S, Act, \Delta, \Delta, s_0)$. An MTS (or LTS) is *deterministic* when no state has more than one outgoing transition on the same action. We refer to transitions in $\Delta^p \backslash \Delta^r$ as *maybe* transitions. In figures, maybe transitions are denoted with a question mark following the label. Example MTSs are shown in Figure 5 and Figure 9 as well as many other figures in this paper.

Definition 4: An *empty* MTS (LTS), denoted $\mathcal{E}$, is the MTS (LTS) with a single state and an empty transition relation.

An MTS $M = (S,\ Act,\ \Delta^r,\ \Delta^p,\ s_0)$ has a required transition on ℓ (denoted $M \xrightarrow{\ell}_r M'$) if $M' = (S,\ Act,\ \Delta^r,\ \Delta^p,\ s_0')$ and $(s_0, \ell, s_0') \in \Delta^r$. Similarly, M has a maybe transition on ℓ (denoted $M \xrightarrow{\ell}_m M'$) if $(s_0, \ell, s_0') \in \Delta^p - \Delta^r$. $M \xrightarrow{\ell}_p M'$ means $(s_0, \ell, s_0') \in \Delta^p$.

Definition 5: (Traces) A *trace* $\pi = a_0, a_1, \ldots$ where $a_i \in Act$ is a *required trace* in an MTS M if there exists a sequence $\{s_i\}$ such that $M_0 = M$ and $M_i \xrightarrow{a_i}_r M_{i+1}$ for all $i \in \mathbb{N}$. A trace π is a *maybe trace* in M if π is not a required trace, but there exists an infinite sequence $\{M_i\}$ such that $M_0 = M$ and $M_i \xrightarrow{a_i}_p M_{i+1}$ for all $i \in \mathbb{N}$. A trace π is a *possible trace* in M if π is a maybe or required trace in M. Finally, a trace π is a *false trace* in M if it is not a possible trace.

We denote the set of required, maybe, possible, and false traces over a given MTS M by $\textsc{ReqTr}(M)$, $\textsc{MaybeTr}(M)$, $\textsc{PosTr}(M)$, and $\textsc{FalseTr}(M)$, respectively. For an LTS $L = (S, Act, \Delta, s_0)$, we denote by $\textsc{Tr}(L)$ the set of required traces of its embedding into MTS, $M = (S, Act, \Delta, \Delta, s_0)$, so that $\textsc{Tr}(L) = \textsc{ReqTr}(M)$. We annotate these sets with "$_\infty$" to mean that they capture only infinite-length traces, e.g., $\textsc{ReqTr}_\infty(M)$ denotes the set of infinite-length required traces of M.

We use *refinement* to capture the notion of elaboration of a partial description into a more comprehensive one, in which some knowledge of the maybe behaviour has been gained. Refinement can be seen as being a "more defined than" relation between two partial models. Intuitively, an MTS N refines M if N preserves all of the required and all of the proscribed behaviours of M. Alternatively, an MTS N refines M if N can simulate the required behaviour of M, and M can simulate the possible behaviour of N.

$$\text{MM}\ \frac{M \xrightarrow{\ell}_m M',\ N \xrightarrow{\ell}_m N'}{M \| N \xrightarrow{\ell}_m M' \| N'} \qquad \text{TT}\ \frac{M \xrightarrow{\ell}_r M',\ N \xrightarrow{\ell}_r N'}{M \| N \xrightarrow{\ell}_r M' \| N'}$$

$$\text{MT}\ \frac{M \xrightarrow{\ell}_m M',\ N \xrightarrow{\ell}_r N'}{M \| N \xrightarrow{\ell}_m M' \| N'}$$

Fig. 6. Rules for parallel composition.

Definition 6: (Refinement) Let $\wp_M$ be the universe of all MTSs. An MTS N_0 is a *refinement* of an MTS M_0, written $M_0 \preceq N_0$, if there exists some refinement relation $R \subseteq \wp_M \times \wp_M$ such that $(M_0, N_0) \in R$ and $\forall \ell \in Act$, $\forall (M, N) \in R$, the following holds:

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

For example, $\mathcal{A}$ in Figure 5 is refined by $\mathcal{B}$ via the relation $\{(0,0), (0,1)\}$ since $\mathcal{A}$ has no required behaviour for $\mathcal{B}$ to simulate, and $\mathcal{A}$ can simulate the possible behaviour of $\mathcal{B}$.

Definition 7: (Equivalence) Models M and N are *equivalent*, written $M \equiv N$, if $M \preceq N$ and $N \preceq M$.

LTSs that refine an MTS M are complete descriptions of the system behaviour and thus are called *implementations* of M. So, an MTS M can be thought of as a model that represents the set of LTSs that implement it, denoted $\mathcal{I}(M)$.

Definition 8: (Implementation) An LTS L is an *implementation* of an MTS M if and only if $M \preceq L$.
For example, the LTS in Figure 3 is an implementation of an MTS in Figure 10.

An implementation is *deadlock free* if all states have outgoing transitions. We refer to the set of deadlock-free implementations of M as $\mathcal{I}_\infty(M)$.

Definition 9: (Deadlock-free Implementation) An LTS $L = (S_L,\ Act,\ \Delta_L,\ s_{0L})$ 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 $a \in Act$ and $s' \in S_L$ such that $L_s \xrightarrow{a} L_{s'}$.

Parallel composition [35] captures the notion of MTSs that run asynchronously but synchronize on shared actions.

Definition 10: (Parallel Composition) Let $M = (S_M, Act, \Delta^r_M, \Delta^p_M, s_{0M})$ and $N = (S_N, Act, \Delta^r_N, \Delta^p_N, s_{0N})$ be MTSs. *parallel composition* ($\|$) is a symmetric operator and $M \| N$ is the MTS $(S_M \times S_N, Act, \Delta^r, \Delta^p, (s_{0M}, s_{0N}))$, where Δ^r and Δ^p are the smallest relations that satisfy the rules in Figure 6.

For example, the parallel composition of MTSs $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{C} = \mathcal{A} \| \mathcal{B}$, is depicted in Figure 5.

B. MTS Merging

Merging MTSs [48] is the process of combining what is known from each partial behaviour description; in other words, it is the construction of an MTS that includes all the required and all the prohibited behaviours from each MTS, and is as least refined as possible. Formally, merging MTSs is the process of finding their minimal common refinement.

Definition 11: (Minimal Common Refinement) Let Q, M, and N be MTSs. Q is a *common refinement (CR)* of M and N if $M \preceq Q$ and $N \preceq Q$. Q is a *minimal common refinement* (MCR) of M and N if Q is a CR of M and N, such that $Q \equiv Q'$ whenever Q' is a CR of M and N, and $Q' \preceq Q$.

That is, a minimal common refinement of two models, when it exists, is a least refined model that contains all information present in either model. Often, there are several MCRs of two partial models.

Definition 12: [34] Let M and N be MTSs. If they have an MCR and it is unique, it is called their *least common refinement* (LCR).

Two MTSs are consistent if and only if they have common deadlock-free implementations:

Definition 13: (Consistency) MTSs M and N are *consistent* if and only if $\mathcal{I}_\infty(M) \cap \mathcal{I}_\infty(N) \neq \emptyset$.

In other words, two MTSs are consistent if there is an MTS, that allows infinite behaviours, that is a common refinement of both.

Although previous work on refinement and merge does not restrict these notions to MTSs with deadlock-free implementations, we introduce this additional constraint to enable us to define the notion of satisfaction of temporal logic over infinite traces (see Section IV and Section X-B).

Given two MTSs M and N that are consistent, it is possible to construct their minimal common refinement using the $+$ operator defined in [11]. Although this operator, in contrast to [31], can be applied to arbitrary consistent models, the models we merge in this paper are *deterministic*, allowing us to use a simpler version of this operator, first defined in [3] and reproduced below. It follows from the proofs in [11] that for a pair of consistent and deterministic MTSs, their least common refinement exists, and the $+$ operator defined below constructs it. The constructed common refinement is based on the *consistency relation*.

Definition 14: (Consistency Relation) [11] Let $M_0 = (S_M, Act, \Delta^r_M, \Delta^p_M, s_{0M})$ and $N_0 = (S_N, Act, \Delta^r_N, \Delta^p_N, s_{0N})$ be MTSs. A *consistency relation* is a binary relation $R \subseteq \Delta^r_M \times \Delta^r_N$, such that the following conditions hold for all $(M, N) \in R$:

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

Theorem 1: (Consistency Implies Consistency Relation) [11] For every pair of consistent MTSs M and N, there exists a consistency relation R such that (M, N) is contained in R.

Definition 15: (The + Operator [11], [3]) Let $M = (S_M, Act, \Delta^r_M, \Delta^p_M, s_{0M})$ and $N = (S_N, Act, \Delta^r_N, \Delta^p_N, s_{0N})$ be deterministic and consistent MTSs. In addition, let C_{MN} be the largest consistency relation such that $(M, N) \in C_{MN}$. Then $+$ is a symmetric operator, and $M + N$ is the MTS $(C_{MN}, Act, \Delta^r, \Delta^p, (s_{0M}, s_{0N}))$, where Δ^r and Δ^p are the smallest relations satisfying the rules in Figure 7.

Theorem 2: [11] For every pair of consistent and deterministic MTSs M and N, the $+$ operator produces their LCR.

The merge operator, $+$, differs from parallel composition in two aspects. First, the state space of the merge is restricted to pairs of states that are consistent, as defined via the consistency relation (Definition 14). Second, merge differs in the rule MT, used when synchronizing a maybe with a required transition (the TT and MM rules for both operators are the same). Merging these transitions results in a required transition instead of a maybe. For example, the

$$\text{MM}\ \frac{M \xrightarrow{\ell}_m M',\ N \xrightarrow{\ell}_m N'}{M+N \xrightarrow{\ell}_m M'+N'} \qquad \text{TT}\ \frac{M \xrightarrow{\ell}_r M',\ N \xrightarrow{\ell}_r N'}{M+N \xrightarrow{\ell}_r M'+N'}$$

$$\text{MT}\ \frac{M \xrightarrow{\ell}_m M',\ N \xrightarrow{\ell}_r N'}{M+N \xrightarrow{\ell}_r M'+N'}$$

Fig. 7. Rules for the + operator.

model $\mathcal{B} = \mathcal{A} + \mathcal{B}$, shown in Figure 5, differs from $\mathcal{C} = \mathcal{A} \| \mathcal{B}$ on the a-transition between states 0 and 1.

The intuition is that knowledge is being added, so when a transition is required in one of the models, it is required in the merge.

IV. 3-valued FLTL

In this paper, we assume that properties are specified using Fluent Linear Temporal Logic (FLTL) [13]. Linear temporal logics are widely used to describe behaviour requirements [13], [52], [25]. The motivation for choosing FLTL is that it provides a uniform framework for specifying and model-checking state-based temporal properties in event-based models [13].

In this section, we first review the 3-valued Kleene logic [26] and use it to define a 3-valued variant of Fluent Linear Temporal Logic (FLTL) [13]. We then prove that FLTL properties are preserved by MTS refinement and define a model-checking procedure for this logic.

A. 3-Valued Logic

The truth values $\mathbf{t}$ (*true*), $\mathbf{f}$ (*false*), and $\bot$ (*maybe*, *unknown*) form the Kleene logic, which we refer to as $\mathbf{3}$. These truth values can have two orderings, $\sqsubseteq$ (truth), which satisfies $\mathbf{f} \sqsubseteq \bot \sqsubseteq \mathbf{t}$, and $\sqsubseteq_{inf}$ (information), which satisfies $\bot \sqsubseteq_{inf} \mathbf{t}$ and $\bot \sqsubseteq_{inf} \mathbf{f}$ (i.e., *maybe* gives the least amount of information), and both orderings are idempotent. 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 $\bot$:

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

B. Models and Implementations

FLTL [13] is a linear-time temporal logic for reasoning about fluents. A fluent *Fl* is defined by a pair of sets I_{Fl}, the set of initiating actions, and T_{Fl}, the set of terminating actions:

$$Fl = \langle I_{Fl}, T_{Fl} \rangle \text{ where } I_{Fl}, T_{Fl} \subseteq Act \text{ and } I_{Fl} \cap T_{Fl} = \emptyset.$$

A fluent may be initially *true* or *false* as indicated by the $Initially_{Fl}$ attribute, where a lack of this attribute indicates that the fluent is initially *false*.

Every action $a \in Act$ induces a fluent, namely, a means $\langle a, Act \setminus \{a\} \rangle$. For example, consider the property p_3 for the Webmail system described in Figure 2. It uses fluents *logout* and *logoutMsg* derived from the actions with the same name and defined as above.

Given the 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:

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

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

Fig. 8. Semantics for the satisfaction operator.

where $Fl \in \Phi$.

Let Π be the set of infinite traces over *Act*. For $\pi \in \Pi$, we write π^i for the suffix of π starting at a_i. π^i satisfies a fluent *Fl*, denoted $\pi \models Fl$, if and only if one of the following conditions holds:

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

In other words, a fluent holds at a time instant if and only if it holds initially, or some initiating action has occurred, and in both cases, 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 8. This definition is standard [13] 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 $\forall \pi \in \Pi \cdot \pi \models \varphi$.

The 3-valued semantics of FLTL over an MTS M is given by the function $\|\cdot\|^M$ that, for each formula $\varphi \in$ FLTL, returns the truth value of φ in M, i.e., $\mathbf{t}$, $\mathbf{f}$ or $\perp$.

We restrict the function to those MTSs M that have at least one deadlock-free implementation, i.e., $\mathcal{I}_\infty(M) \neq \emptyset$. With this restriction, when M is an LTS, our 3-valued semantics reduces to the standard 2-valued semantics of FLTL.

Assumption: In the remainder of the paper, we consider only those MTSs which have at least one deadlock-free implementation.

If a property evaluates to *true* in M, it is *true* in all deadlock-free implementations of M, and if a property evaluates to *false* in M, it is *false* in all deadlock-free implementations of M. Furthermore, if a property evaluates to *maybe* in M, it is *true* in some deadlock-free implementations of M and *false* in others. This semantics of a 3-valued logic is referred to as *thorough* [4].

Definition 16: (3-valued Semantics of FLTL) The function $\|\cdot\|^M : FLTL \longrightarrow \mathbf{3}$ is defined as follows:

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

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

For example, a formula $\Diamond a$ is *true* in $\mathcal{B}$ (see Figure 5) because every trace in $\textsc{PosTr}_\infty(\mathcal{B})$ satisfies $\Diamond a$, whereas $\Box b$ is *false* in $\mathcal{B}$ because every trace in $\textsc{PosTr}_\infty(\mathcal{B})$ refutes $\Box b$. Finally, $\Diamond b$ is *maybe* in $\mathcal{A}$: it is not *true* in $\mathcal{A}$ because not all possible traces satisfy $\Diamond b$ (e.g., $a, a, \ldots$ does not); it is not *false* in $\mathcal{A}$ because there is no required trace that refutes the property and there is a possible trace that does not refute it (e.g. $b, b, \ldots$).

The information ordering of this 3-valued variant of FLTL is preserved by refinement, i.e., refinement preserves all *true* and *false* properties.

Definition 17: For an MTS M and a trace π in it, let $M(\pi)$ be the *value of* π *in* M, defined as follows: $M(\pi) = \mathbf{t}$ iff $\pi \in \textsc{ReqTr}(M)$, $M(\pi) = \perp$ iff $\pi \in \textsc{PosTr}(M) \wedge \pi \notin \textsc{ReqTr}(M)$. $M(\pi) = \mathbf{f}$, otherwise.

We begin with the following lemma which states that the information ordering on the values of traces is preserved under refinement.

Lemma 1: (Preservation of Trace Values) Let M and N be MTSs over the same vocabulary. Then,

$$M \preceq N \Rightarrow \forall \pi \in \Pi \cdot M(\pi) \sqsubseteq_{inf} N(\pi)$$

Proof: Let $\pi = a_0, a_1, a_2 \ldots$ and let $M(\pi) = \mathbf{t}$. By Definition 17, there exists a sequence $\{M_i\}$ such that $M = M_0$ and $M_i \xrightarrow{a_i}_r M_{i+1}$ for all $i \in \mathbb{N}$. Since $M \preceq N$, there exists a sequence $\{N_i\}$ such that $N = N_0$ and $N_i \xrightarrow{a_i}_r t_{N+1}$ for all $i \in \mathbb{N}$. Therefore, $N(\pi) = \mathbf{t}$ by Definition 17.

Let $\pi = a_0, a_1, a_2, \ldots$ and let $M(\pi) = \mathbf{f}$. Suppose that $N(\pi) = \perp$. Then π is a possible trace in N. Therefore, by Definition 17, there exists a sequence $\{N_i\}$ such that $N_0 = N$ and $N_i \xrightarrow{a_i}_p N_{i+1}$ for all $i \in \mathbb{N}$. Since $M \preceq N$, there exists a sequence $\{M_i\}$ such that $M = M_0$ and $M_i \xrightarrow{a_i}_p M_{i+1}$ for all $i \in \mathbb{N}$. By Definition 17, $M(\pi) = \perp$, contradicting $M(\pi) = \mathbf{f}$. Therefore, $N(\pi) = \mathbf{f}$. ■

Stating Lemma 1 in terms of sets of traces, we obtain the following corollary:

Corollary 1: Let M and N be MTSs over the same vocabulary, and let $M \preceq N$. Then, $\textsc{ReqTr}(M) \subseteq \textsc{ReqTr}(N)$, $\textsc{FalseTr}(M) \subseteq \textsc{FalseTr}(N)$, and $\textsc{MaybeTr}(M) \supseteq \textsc{MaybeTr}(N)$.

The above lemma and its corollary are fundamental for proving preservation of any linear time logic property.

We are now ready to state the preservation theorem.

Theorem 3: (Preservation of FLTL) Let M and N be MTSs such that $M \preceq N$. Then, $\forall \varphi \in FLTL \cdot \|\varphi\|^M \sqsubseteq_{inf} \|\varphi\|^N$.

Proof: Let $\varphi \in$ FLTL. We do a proof by contradiction for value $\mathbf{t}$.

$$\begin{array}{ll}
 & (\|\varphi\|^M = \mathbf{t}) \wedge \neg(\|\varphi\|^N = \mathbf{t}) \\
 & \text{(Definition 16 on negation of 1st conjunct)} \\
= & (\|\varphi\|^M = \mathbf{t}) \wedge \exists \pi \in \textsc{PosTr}_\infty(N) \cdot \pi \not\models \varphi \\
 & \text{(Definition 16 on 1st conjunct)} \\
= & \forall \pi \in \textsc{PosTr}_\infty(M) \cdot \pi \models \varphi \wedge \exists \pi \in \textsc{PosTr}_\infty(N) \cdot \pi \not\models \varphi \\
 & \text{(Corollary 1 and Law of Contradiction)} \\
= & \mathbf{f}
\end{array}$$

Therefore, from above and the definition of refinement, if

$(\|\varphi\|^M = \mathbf{t})$ then $(\|\varphi\|^N = \mathbf{t})$.

$$\begin{aligned}
& \|\varphi\|^M = \mathbf{f} \\
& \text{(Definition 16)} \\
= \quad & (\exists \pi \in \textsc{ReqTr}_\infty(M) \cdot \pi \not\models \varphi) \\
& \quad \vee (\forall \pi \in \textsc{PosTr}_\infty(M) \cdot \pi \not\models \varphi) \\
& \text{(Corollary 1)} \\
\Rightarrow \quad & (\exists \pi \in \textsc{ReqTr}_\infty(N) \cdot \pi \not\models \varphi) \\
& \quad \vee (\forall \pi \in \textsc{PosTr}_\infty(N) \cdot \pi \not\models \varphi) \\
& \text{(Definition 16)} \\
= \quad & \|\varphi\|^N = \mathbf{f}.
\end{aligned}$$

■

Definition 18: An FLTL formula φ is *satisfiable* if and only if there exists a (deadlock-free) LTS L such that $L \models \varphi$; otherwise, φ is *unsatisfiable*.

For example, no deadlock-free LTS satisfies $a \wedge \neg a$, and thus it is unsatisfiable.

Note that instead of requiring non-deadlocking implementations in order to define FLTL over infinite traces, we could have defined the satisfaction operator to allow for finite traces. However, this brings the issue of preserving properties containing disjunction through refinement (see Section X-B for more detail).

C. Model-Checking

Let M^+ be the LTS obtained from an LTS M by converting all maybe transitions to required and then removing all transitions that are not part of an infinite trace and all states that are not reachable from the initial state. It is easy to show that $\textsc{Tr}(M^+)$=$\textsc{PosTr}_\infty(M)$. Similarly, let M^- be the LTS obtained from converting all maybe transitions to false and then removing all transitions that are not part of an infinite trace and all states that are not reachable from the initial state. M^- represents the required traces of M, i.e., $\textsc{Tr}(M^-)$=$\textsc{ReqTr}_\infty(M)$. Recall (see Definition 4) that the LTS M with an empty set of traces (i.e., $\textsc{Tr}(M) = \emptyset$) is the empty LTS, denoted $\mathcal{E}$.

Theorem 4: (Model-checking) For every MTS M such that $\mathcal{I}_\infty(M) \neq \emptyset$, the following holds:

1. $\|\varphi\|^M = \mathbf{t} \Leftrightarrow M^+ \models \varphi$
2. $\|\varphi\|^M = \mathbf{f} \Leftrightarrow \begin{cases} M^+ \models \neg\varphi, \text{ if } M^- \equiv \mathcal{E} \\ M^- \not\models \varphi, \text{ otherwise} \end{cases}$
3. $\|\varphi\|^M = \perp \Leftrightarrow \|\varphi\|^M \neq \mathbf{t} \ \wedge \ \|\varphi\|^M \neq \mathbf{f}$.

Proof: Consider condition 1. $\|\varphi\|^M = \mathbf{t}$ means that all traces in $\textsc{PosTr}_\infty(M)$ satisfy φ. Since we assume that M has deadlock-free implementations, $M^+ \not\equiv \mathcal{E}$. As a consequence, $M^+ \models \varphi$ iff $\textsc{PosTr}_\infty(M)$=$\textsc{Tr}(M^+)$.

Consider condition 2, from left to right. If $\|\varphi\|^M = \mathbf{f}$, then either there is an infinite required trace of M that refutes φ, or all infinite possible traces of M refute φ. If π is a required infinite trace of M that refutes φ, then $\textsc{ReqTr}_\infty(M) \neq \emptyset$ and $M^- \not\equiv \mathcal{E}$. In addition, $\pi \in \textsc{Tr}(M^-)$ which entails that $M^- \not\models \varphi$. If $\forall \pi \in \textsc{PosTr}_\infty(M) \cdot \pi \not\models \varphi$, then either one of these is a required trace, which takes us to the previous case, or there are no infinite required traces, in which case $M^- \equiv \mathcal{E}$. Given that all infinite traces of M refute φ and that $\textsc{PosTr}_\infty(M)$=$\textsc{Tr}(M^+)$, we have $M^+ \models \neg\varphi$.

Consider condition 2, from right to left. We first assume that $M^- \equiv \mathcal{E}$ and that $M^+ \models \neg\varphi$. This means that all traces of M^+ refute φ. Because $\textsc{PosTr}_\infty(M)$=$\textsc{Tr}(M^+)$, all infinite possible traces of M refute φ, which by Definition 16 means that $\|\varphi\|^M = \mathbf{f}$. We now assume that $M^- \not\equiv \mathcal{E}$ and that $M^- \not\models \varphi$. Then there is a trace in $\pi \in Tr(M^-)$ that refutes φ. As $\textsc{ReqTr}_\infty(M)$=$\textsc{Tr}(M^-)$, there is an infinite positive trace of M that refutes φ, which by Definition 16 means that $\|\varphi\|^M = \mathbf{f}$. ■

The above theorem shows that 3-valued FLTL model-checking essentially reduces to classical FLTL model-checking, and thus can be supported by the LTSA tool [40] (see Section VIII). Such a reduction has already been specified for *branching-time* temporal logics and partial models of Kripke structures in [4].

Note that in the above theorem, we assumed that M has deadlock-free implementations. This assumption can be checked by simply constructing M^+ and checking that it is not the empty LTS.

V. Synthesis from Properties

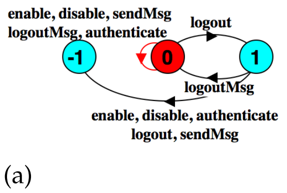
In this section, we describe and analyze an algorithm for synthesizing an MTS for a safety property given as an FLTL formula. Safety properties are those that specify that “nothing bad can happen” and that can be falsified by a finite sequence of events. In addition, we prove that the algorithm builds an MTS that characterizes all deadlock-free LTS models that satisfy the property used for synthesis. The algorithm is an extension of an existing algorithm for synthesizing LTSs [37], which is based on the algorithm for Büchi automaton construction from FLTL properties, given in [13]. We first present the LTS synthesis algorithm from [37] and discuss its limitations, and then extend it to synthesize MTSs. Finally, we prove correctness,completeness and a number of useful properties of the synthesis algorithm. We leave to Section X the discussion on the ramifications of restricting our approach to safety properties.

A. LTS Synthesis from Properties

The technique for model-checking an FLTL property φ over an LTS L involves constructing a Büchi automaton $B(\neg\varphi)$ that recognizes all infinite traces over the alphabet *Act* that violate φ and checking that the synchronous product of $B(\neg\varphi)$ with L is empty [13]. This method is based on the treatment of ordinary LTL [53].

In this paper, we fix the alphabet to be the universe of actions of the system under construction, namely, *Act*, so that the synthesis procedure always produces models with the same alphabet, regardless of whether the actions appear in φ (see Section X for a discussion on this).

As pointed out in [37], when φ is a safety property, $B(\neg\varphi)$ has only one accepting state. All of the outgoing transitions from this state are self-loop transitions, reflecting the fact that safety properties are violated by a finite sequence of actions and that this violation cannot be remedied. Thus, $B(\neg\varphi)$ can be viewed as a *property* LTS for φ, i.e., an LTS with an error state which corresponds to the accepting state of $B(\neg\varphi)$. All traces that reach the error state correspond



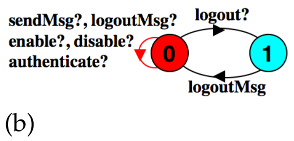


Fig. 9. (a) the property LTS for p_3; (b) the MTS $M(p_3)$.

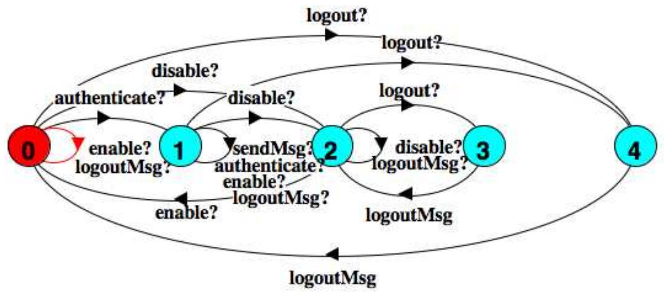


Fig. 10. $M(p_1 \wedge p_2 \wedge p_3)$ or $M(p_1) + M(p_2) + M(p_3)$.

to undesired behaviours, i.e., no infinite trace with a finite suffix that leads to the error state satisfies φ.

For example, the property LTS for p_3 of the Webmail system, which states that a logout message must be sent to the user as soon as it logs out, is shown in Figure 9(a), where the error state is denoted by -1. In this LTS, the trace *logout*, *authenticate* is illegal (it leads to state -1), so no infinite trace starting with *logout*, *authenticate* satisfies p_3. For details on constructing a property LTS, see [13].

Once the property LTS has been constructed, all transitions not corresponding to an infinite trace are removed. Subsequently, all states that are unreachable from the initial state are removed. This last step is guaranteed to remove at least the error state. The resulting system is an LTS that captures all infinite traces on the system alphabet that satisfy φ, because the property LTS contains all infinite traces over the alphabet that do not violate φ, and the finite traces that are removed correspond to all infinite traces in $B(\neg\varphi)$ that refute φ. We denote by $L(\varphi)$ the LTS generated by this procedure (e.g., $L(p_3)$ is the LTS in Figure 9(a) with state -1 removed). Note that by construction, $L(\varphi)$ is *deterministic* and deadlock free.

An automaton for an unsatisfiable property φ does not have any infinite traces, since they refute φ. Since all finite traces in the property LTS are removed, $L(\varphi) \equiv \mathcal{E}$.

Theorem 5: (LTS parallel composition is conjunction) [1] If φ_1 and φ_2 are safety FLTL properties over the same universe of actions Act such that $\varphi_1 \wedge \varphi_2$ is satisfiable, then

$$L(\varphi_1) \| L(\varphi_2) \equiv L(\varphi_1 \wedge \varphi_2).$$

B. MTS Synthesis from Properties

In order to overcome the limitations described in Section II, we extend the synthesis procedure for LTSs to synthesize an MTS from a safety property φ, expressed in FLTL. The algorithm is called `MTSprop`:

1) let $L = L(\varphi)$ (constructed as described in Section V-A);
2) return $M(\varphi)$, where $M(\varphi)$ is the MTS obtained from L by converting all outgoing transitions for each $s \in S_L$ to maybe transitions, whenever s has more than one outgoing transition.

For a satisfiable safety property φ, $L(\varphi)$ contains all infinite traces that satisfy φ and no traces that refute φ. When a state in $L(\varphi)$ has more than one outgoing transition, there is more than one way to satisfy φ at that point in the trace. Thus, not all such transitions are necessary to satisfy φ, but any LTS that satisfies φ must contain at least one of them. Such choices should be modelled with maybe instead of required behaviour, as in step 2 of `MTSprop`. Note that although a valid refinement of the synthesized MTS could remove all the outgoing maybe transitions from a state, we will not be considering such refinements; instead, we restrict our results to deadlock-free implementations of the synthesized MTS. Along the same lines, if a state has only one outgoing transition, then this transition should be required because we are interested in deadlock-free implementations.

For example, $M(p_3)$ is shown in Figure 9(b). The only required behaviour in this system is from state 1 to state 0 on action *logoutMsg*, because this is the only event required by this property. A possible refinement of $M(p_3)$ is $M(P)$ depicted in Figure 10 (i.e., $M(p_3) \preceq M(P)$ via the refinement relation $\{(0,0),(0,1),(0,2),(1,3),(1,4)\}$). Following the discussion of Section II, note that $M(P)$ distinguishes required from maybe *logoutMsg* transitions while $L(P)$ of Figure 3 does not.

We now study properties of the algorithm `MTSProp`, including correctness (all deadlock-free implementations of $M(\varphi)$, the result of the algorithm, satisfy φ) and completeness (all deadlock-free LTSs that satisfy φ are refinements of $M(\varphi)$).

We begin by showing that φ holds in the MTS $M(\varphi)$ computed by the algorithm.

Lemma 2: For every satisfiable safety property φ,

$$\|\varphi\|^{M(\varphi)} = \mathbf{t}.$$

Proof: $L(\varphi)$ contains all traces that satisfy φ and no traces that refute φ. Therefore, every trace in $\text{PosTr}(M(\varphi))$ satisfies φ and so $\|\varphi\|^{M(\varphi)} = \mathbf{t}$ by Definition 16. ■

Lemma 2 is essential for proving correctness of `MTSprop`, namely, that all deadlock-free implementations of $M(\varphi)$ satisfy φ:

Theorem 6: (Correctness of `MTSprop`*)* For every satisfiable safety property φ and every LTS $L \in \mathcal{I}_\infty(M(\varphi))$, $L \models \varphi$.

Proof: Follows from Lemma 2 and Theorem 3. ■

Since `MTSprop` is effectively doing an LTL tableaux construction [39], its complexity is exponential in the size of the FLTL formula and linear in the size of the global MTS.

We now turn our attention to proving completeness of the algorithm `MTSprop`, and begin by showing that every MTS satisfying φ is a refinement of $M(\varphi)$.

Lemma 3: For every satisfiable safety property φ,

$$\forall M \in \wp_M \cdot \|\varphi\|^M = \mathbf{t} \Rightarrow M(\varphi) \preceq M.$$

Proof: We show how to build a refinement relation R between $M(\varphi)$ and M. Let $(M(\varphi), M) \in R$, i.e., initial states are related. Consider any $(M(\varphi)', M') \in R$. Suppose

$M(\varphi)' \xrightarrow{a}_r M(\varphi)''$ for some $a \in Act$. Then $M(\varphi)' \overset{b}{\not\longrightarrow}$ for any $b \neq a$ by step 2 of `MTSprop`. Since $\|\varphi\|^{M'} = \mathbf{t}$, by Definition 16, $\forall \pi \in \text{PosTr}(M') \cdot \pi \models \varphi$. Thus, every π must start with a; otherwise, $L(\varphi)$ would have more than one outgoing transition, and hence so would $M(\varphi)'$. Thus, $\exists M'' \cdot M' \xrightarrow{a}_r M''$, so $(M(\varphi)'', M'') \in R$. Therefore, Condition (1) in Definition 6 holds.

$M(\varphi)$ contains all traces that satisfy φ, and therefore $\text{PosTr}(M) \subseteq \text{PosTr}(M(\varphi))$. In addition, for an action a_i in π, $\exists M(\varphi)_i, M(\varphi)_{i+1} \cdot M(\varphi)_i \xrightarrow{a_i}_p M(\varphi)_{i+1}$. Additionally, there is no $b \neq a_i$ such that $M(\varphi)_i \xrightarrow{b}_r$, because step 2 of algorithm `MTSprop` would have converted it to a maybe transition. Thus, the sequence of states in M corresponding to π can be identified in R with the corresponding sequence of states in $M(\varphi)$, and therefore Condition (2) in Definition 6 holds. ■

We are ready to prove completeness of `MTSprop`: all deadlock-free LTSs that satisfy φ are refinements of $M(\varphi)$, formalized as follows:

Theorem 7: (Completeness of `MTSprop`*)* For every satisfiable safety property φ,

$$\forall L \in \text{deadlock-free LTSs} \cdot L \models \varphi \Rightarrow M(\varphi) \preceq L.$$

Proof: Follows from Lemma 3 and the fact that 3-valued FLTL semantics over MTSs reduces to 2-valued FLTL semantics over LTSs (see Section IV). ■

Hence, by Theorems 6 and 7, given a safety property φ, the synthesis procedure described above characterizes all MTSs and deadlock-free LTSs that satisfy φ. Formally,

Corollary 2: (Characterization of φ) If φ is a satisfiable safety property, then $\forall M \in \wp_M \cdot M \models \varphi \Leftrightarrow M(\varphi) \preceq M$. In particular, for LTSs L, $L \models \varphi \Leftrightarrow L \in \mathcal{I}_\infty(M(\varphi))$.

The practical implication of the above theorem is that the synthesis procedure effectively constructs an MTS from which *all* possible system models that satisfy the given properties can be reached through the elaboration of the maybe behaviour. For example, recall from Section II that the LTS model, $L(P)$, of the Webmail system cannot be refined to a model which requires the trace sc_2. In contrast, the MTS $M(P)$ supports this refinement by replacing the maybe transitions $0 \xrightarrow{authenticate}_m 1$, $1 \xrightarrow{sendMsg}_m 1$, $1 \xrightarrow{disable}_m 2$, $2 \xrightarrow{logoutMsg}_m 2$, with the required transitions.

We now prove two theorems that relate property synthesis, merge and 3-valued FLTL semantics. First, we show that consistency of MTSs synthesized from two properties is equivalent to satisfiability of these properties.

Theorem 8: (Consistency) $M(\varphi_1)$ and $M(\varphi_2)$ are consistent if and only if $\varphi_1 \wedge \varphi_2$ is satisfiable.

Proof: From left to right: If $M(\varphi_1)$ and $M(\varphi_2)$ are consistent then there is a deadlock-free LTS such that $M(\varphi_1) \preceq L$ and $M(\varphi_2) \preceq L$. Thus, $L \models \varphi_1 \wedge \varphi_2$, and $\varphi_1 \wedge \varphi_2$ is satisfiable.

From right to left: If $\varphi_1 \wedge \varphi_2$ is satisfiable, then there is a deadlock-free LTS L such that $L \models \varphi_1 \wedge \varphi_2$. Hence, $L \models \varphi_1$ and $L \models \varphi_2$. From Corollary 2, $M(\varphi_1) \preceq L$ and $M(\varphi_2) \preceq L$. Hence, from Definition 13, $M(\varphi_1)$ and $M(\varphi_2)$ are consistent. ■

Theorem 8 provides a procedure for determining the satisfiability of $\varphi_1 \wedge \varphi_2$. It also provides the basis for logical conjunction, because if $\varphi_1 \wedge \varphi_2$ is satisfiable and $M(\varphi_1)$ and $M(\varphi_2)$ are consistent, then, given that both MTS are deterministic, their merge, $M(\varphi_1) + M(\varphi_2)$, is guaranteed to exist and be unique (see Theorem 2). Therefore, for a satisfiable safety property $\varphi_1 \wedge \varphi_2$, building the MTS for $\varphi_1 \wedge \varphi_2$ using the algorithm `MTSprop` is equivalent to building the MTSs for φ_1 and φ_2 individually and then merging them, as formalized below.

Theorem 9: (Conjunction) For every satisfiable FLTL safety property $\varphi_1 \wedge \varphi_2$, $M(\varphi_1 \wedge \varphi_2) \equiv M(\varphi_1) + M(\varphi_2)$.

Proof: $\|\varphi_1\|^{M(\varphi_1 \wedge \varphi_2)} = \mathbf{t}$ and $\|\varphi_2\|^{M(\varphi_1 \wedge \varphi_2)} = \mathbf{t}$ implies that $M(\varphi_1) \preceq M(\varphi_1 \wedge \varphi_2)$ and $M(\varphi_2) \preceq M(\varphi_1 \wedge \varphi_2)$ by Theorem 7. Therefore $M(\varphi_1 \wedge \varphi_2)$ is a common refinement of $M(\varphi_1)$ and $M(\varphi_2)$, and thus $M(\varphi_1) + M(\varphi_2) \preceq M(\varphi_1 \wedge \varphi_2)$ by Definition 15.

By the previous argument, $M(\varphi_1)$ and $M(\varphi_2)$ are consistent and $\varphi_1 \wedge \varphi_2$ is satisfiable. Thus $\|\varphi_1 \wedge \varphi_2\|^{M(\varphi_1)+M(\varphi_2)} = \mathbf{t}$ by Theorem 3 and Definition 15. By Theorem 7, $M(\varphi_1 \wedge \varphi_2) \preceq M(\varphi_1) + M(\varphi_2)$. ■

Theorem 9 is the 3-valued counterpart to Theorem 5.

Putting Theorems 8 and 9 together, we conclude that merging in the realm of MTSs that have at least one deadlock-free implementation is precisely the logical conjunction. Hence, as depicted in Figure 10, building the MTS for the property $p_1 \wedge p_2 \wedge p_3$ is equivalent to building individual MTSs $M(p_1)$, $M(p_2)$ and $M(p_3)$ and merging them.

VI. Synthesis from Scenarios

In this section, we describe an algorithm for synthesizing MTS models from scenario-based specifications. A number of alternative scenario notations, semantics and synthesis techniques exist [49], [29], [27], each with its own advantages and disadvantages. However, the discussion and results presented in this section are not specific to any particular existing approach, and the MTS synthesis algorithm we provide can be used in combination with many of the existing LTS synthesis approaches. The only requirement is that the semantics for the scenario-based description be existential, i.e., that scenarios describe behaviour that the system is expected to exhibit, as opposed to universal properties that all system traces are expected to satisfy. To ground our presentation and provide concrete examples, we use a syntactic subset of the Message Sequence Charts from the ITU standard [24] and the synthesis algorithm presented in [49].

A. LTS Synthesis from Scenarios

The semantics of a scenario-based specification σ can be thought of as a set of traces, i.e., sequences of messages that system components exchange, referred to as $\text{Tr}(\sigma)$.

The requirements for LTS synthesis from a scenario-based specification can vary depending on the assumptions that are made. However, a basic requirement is that the synthesized LTS must be capable of exhibiting the set of traces that are described by the scenarios.

Definition 19: (Soundness of LTS Synthesis from Scenarios) An LTS $L(\sigma)$ is *sound* with respect to a scenario specification σ if and only if $\text{Tr}(\sigma) \subseteq \text{Tr}(L(\sigma))$.

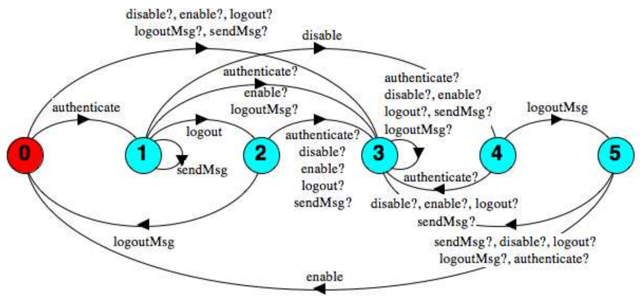


Fig. 11. The MTS $M(sc)$.

For example, the synthesis algorithm described in [49] constructs a sound deterministic LTS model for each component of the scenarios. Each LTS can exhibit exactly the sequence of message exchanges that occur by following the vertical line of the component modelled by this LTS. For example, the LTS for the Server component synthesized from the scenario sc in Figure 1 is shown in Figure 4. Finally, once LTSs for all components have been synthesized, an LTS for the entire system is obtained by composing them in parallel. In the Webmail example, the System LTS is equivalent to that of the Server component.

B. MTS Synthesis from Scenarios

We now provide a synthesis algorithm `MTSscen` that constructs an MTS $M(\sigma)$ from a scenario specification σ. A precondition for this algorithm is the existence of a synthesis algorithm that constructs an LTS $L(\sigma)$ that is sound w.r.t. a scenario specification σ.

1) let $M(\sigma) = L(\sigma)$;
2) add a new state *sink* to $M(\sigma)$ and looping transitions *sink* $\xrightarrow{a}_m$ *sink* for every label $a \in Act$;
3) for every state s in $M(\sigma)$ such that there is no outgoing transition $s \xrightarrow{a}_r$, add $s \xrightarrow{a}_m$ *sink* to $M(\sigma)$;
4) return $M(\sigma)$.

`MTSscen` extends $L(\sigma)$ by turning all traces not explicitly described by σ into maybe traces. It does so by adding a sink state to which all events disallowed by $L(\sigma)$ lead. For instance, $L(sc)$ of Figure 4 is converted into $M(sc)$ of Figure 11, where state 2 is the sink state.

The running time of this algorithm is linear in the number of states in $L(\sigma)$ and the number of labels in Act.

We now show that the MTS synthesized from a scenario specification σ is refined by the LTS synthesized from σ, i.e., $M(\sigma) \preceq L(\sigma)$, and that its required traces subsume the traces specified by σ:

Theorem 10: (Correctness of `MTSscen`*)* Let σ be a scenario specification and $L(\sigma)$ be a deadlock-free LTS sound w.r.t. σ. Furthermore, let $M(\sigma)$ be the result of algorithm `MTSscen` given $L(\sigma)$. Then, $\text{TR}(\sigma) \subseteq \text{REQTR}(M(\sigma))$.

Proof: Let $M_i(\sigma)$ be the MTS resulting from step i of `MTSscen` with $M(\sigma) = M_4(\sigma)$. Step 1 of the algorithm establishes $\text{TR}(L(\sigma)) = \text{REQTR}(M_1(\sigma))$, and steps 2-4 do not add required transitions. Thus, $\text{TR}(L(\sigma)) = \text{REQTR}(M_2(\sigma)) = \text{REQTR}(M_3(\sigma)) = \text{REQTR}(M_4(\sigma))$. Finally, because $L(\sigma)$ is sound w.r.t. σ, we conclude that $\text{TR}(\sigma) \subseteq \text{REQTR}(M(\sigma))$. ■

More importantly, we can show that $M(\sigma)$ characterizes all models that require at least the traces of $\text{TR}(L(\sigma))$. In other words, if the LTS synthesis algorithm guarantees that $\text{TR}(\sigma) = \text{TR}(L(\sigma))$, then refining $M(\sigma)$ guarantees preservation of the scenarios, and *every* implementation that preserves the behaviour of $L(\sigma)$ can be reached by refining $M(\sigma)$.

Theorem 11: (Characterization of σ) Let σ be a scenario specification, $L(\sigma)$ and LTS such that $\text{TR}(\sigma) = \text{TR}(L(\sigma))$, and $M(\sigma)$ be the corresponding result of `MTSscen`. Then, for all LTSs L, $L \in \mathcal{I}(M(\sigma))$ if and only if $L(\sigma) \leq L$.

Proof: From left to right: if $L \in \mathcal{I}(M)$, then there is a refinement relation R such that $(M(\sigma), L) \in R$. In addition, $L(\sigma) \in \mathcal{I}(M(\sigma))$ with refinement relation $R' = \{(M(\sigma)_s, L_s) \mid s \text{ is a state of } L \text{ and } M\ \}$. We now show that $S = \{(A, B) \mid \exists M \cdot (M, A) \in R' \wedge (M, B) \in R\}$ is a simulation relation and that $(L(\sigma), L) \in S$. The latter holds as $(M(\sigma), L) \in R \wedge (M(\sigma), L(\sigma)) \in R'\}$. Let $(A, B) \in S$ and $A \xrightarrow{\ell} A'$. From definition of S we have $M \cdot (M, A) \in R' \wedge (M, B) \in R$, and consequently, $M \xrightarrow{\ell}_p M'$ such that $(M', A') \in R'$. In addition, because R' requires the initial states of M' and A' to be the same, from step 1 of `MTSscen` it must be the case that $M \xrightarrow{\ell}_r M'$. As $(M, B) \in R$, then there exists B' such that $B \xrightarrow{\ell} B'$ and $(M', B') \in R$. Hence, by construction, $(A', B') \in S$. ■

Returning to the Webmail system, the MTS of Figure 11 characterizes all LTS models that are capable of exhibiting at least the scenario sc. All other system behaviours are possible. The addition of safety properties would result in the removal of some of the possible transitions of the MTS, to capture the fact that such behaviours are not allowed, as we show in Section VII.

VII. Synthesis From Properties and Scenarios

In this section, we discuss how to synthesize behaviour models from both safety properties and scenarios. The process consists of merging together a model synthesized from safety properties, described in Section V, and a model synthesized from scenarios, described in Section VI. Key to this process is Theorem 2: the merge operator builds the least common refinement of two MTS models when they are consistent, deterministic and have the same alphabet. In other words, the merge operator builds a model that preserves the required (by scenarios) and the proscribed (by properties) behaviours of the MTSs being composed. Intuitively, both the upper and the lower bounds of the intended system behaviour are preserved by merge; furthermore, both bounds are captured in the same merged MTS model. Note that Theorem 2 is applicable since the synthesis procedures described in Sections V and VI result in deterministic models that have the same alphabet. Furthermore, the merge yields a deterministic MTS as well, allowing composing results of a merge with other synthesized models.

The fact that $+$ builds the least common refinement of the original MTSs (Theorem 2) together with Theorem 12 below means that merging prunes the space of acceptable implementations of two partial descriptions given as MTSs to exactly those that satisfy both partial descriptions. Furthermore, given that an MTS synthesized from a property

sc_5	*enable?*, ...
sc_6	*authenticate*, *logout*, *logoutMsg*, *enable?*, ...
sc_7	*disable?*, *enable?*, *authenticate?*, ...
sc_8	*disable?*, *logoutMsg?*, ...
sc_9	*logout?*, ...
sc_{10}	*disable?*, *disable?*, ...

Fig. 13. Maybe traces of $M_{both} + M(p_4)$.

characterizes all deadlock-free implementations that satisfy the property (Theorem 7) and that an MTS synthesized from a scenario description characterizes all deadlock-free implementations that preserve the scenarios (Theorem 11), we can conclude that the merge operation characterizes all deadlock-free implementations that satisfy the properties used for synthesis and that capture the scenarios used for synthesis.

Theorem 12: If MTSs M and N have the LCR Q, then $\mathcal{I}_\infty(Q) = \mathcal{I}_\infty(M) \cap \mathcal{I}_\infty(N)$.

Proof: Let $L \in \mathcal{I}_\infty(Q)$. As Q is the LCR of M and N, then $M \preceq Q$ and $N \preceq Q$. Given that $L \in \mathcal{I}_\infty(Q)$ implies $Q \preceq L$ and that the operator $\preceq$ is transitive, $L \in \mathcal{I}_\infty(M) \cap \mathcal{I}_\infty(N)$.

Let $L \in \mathcal{I}_\infty(M) \cap \mathcal{I}_\infty(N)$. Since $M \preceq L$ and $N \preceq L$, L is a common refinement. As Q is the least common refinement of M and N, it must be the case that $Q \preceq L$ and thus $L \in \mathcal{I}_\infty(Q)$. ■

The running time of the algorithm for constructing the consistency relation for MTSs $M = (S_M, Act, {\Delta^r}_M, {\Delta^p}_M, s_{0M})$ and $N = (S_N, Act, {\Delta^r}_N, {\Delta^p}_N, s_{0N})$ is $O(m \times n^4 \times log(n))$, while its space complexity is $O(n^2)$, where $n = max(|S_M|, |S_N|)$ and $m = max({\Delta^p}_M, {\Delta^p}_N)$ [11]. When M and N are deterministic, the algorithm for constructing $M + N$ is $O(|R| \times |Act|)$, where R is the consistency relation between M and N.

We now illustrate the utility of these results using the running Webmail example. The MTS that results from merging the MTSs synthesized from properties and scenarios ($M_{both} = M(P) + M(sc)$) is depicted in Figure 12. This MTS characterizes all implementations that are deadlock free, satisfy property P and capture the scenarios described in sc. Further, it can be used to reason about the remaining maybe behaviour, that is, behaviour that does not violate safety property P but has not been explored in the scenario description sc. Consider the maybe trace sc_4 = *authenticate*, *sendMsg*, *authenticate?*, ... of M_{both}. This behaviour is not included in the Webmail scenario specification sc but does not violate the system property P either. Scenario sc_4 may prompt elicitation of a missing precondition for the *authenticate* action: "A user can only be authenticated if he is not already logged in", formalized as

$$p_4 = \Box(\mathbf{X}\ authenticate \Rightarrow !LoggedIn)$$

By construction, the result of merging deterministic MTSs is deterministic, and thus we can apply Theorem 2 to build the minimal common refinement of M_{both} and $M(p_4)$. Furthermore, this reasoning can be used to iteratively merge in new MTSs synthesized from elicited scenarios and properties.

Consider the selection of maybe traces of $M_{both} + M(p_4)$ shown in Figure 13. These traces are not included in the Webmail scenario specification sc and do not violate the system properties P or p_4.

We now hypothesize the decisions that may be made when validating these scenarios with a stakeholder to illustrate how explicit modeling of possible but not required behaviour can help elicit requirements and reason about the system behaviour.

Scenarios sc_5 and sc_6 may elicit a precondition (p_5) for the action *enable*: "A user can only be enabled if he was currently disabled", formalized as

$$p_5 = !enable \wedge \Box(\mathbf{X}\ enable \Rightarrow !Registered)$$

while sc_7 may be identified as a required behaviour, i.e., the system should be capable of allowing users to be disabled before they get authenticated and gain access to the system. In this case, a new scenario could be elicited and added to the existing scenario specification sc, yielding sc' (Figure 14). Note that having an operational model allows us to elicit such scenarios by "walking" the model and guarantees that the new scenarios will satisfy existing safety properties. In our example, the scenario is obtained from the merged MTS by walking through states 0, 1, 3, 4, ...

On the other hand, scenario sc_8 may prompt a more complex property requiring *logoutMsg* to be sent only if the user logs out or is disabled while being logged in:

$$p_6 = !logoutMsg \wedge \Box(\mathbf{X}\ logoutMsg \Rightarrow (logout \vee disable)),$$

whereas sc_9 and sc_{10} may prompt missing preconditions for actions *disable* and *logout*:

$$\begin{aligned} p_7 &= \Box(\mathbf{X}\ disable \Rightarrow Registered) \\ p_8 &= !logout \wedge \Box(\mathbf{X}\ logout \Rightarrow LoggedIn) \end{aligned}$$

Figure 15 depicts a new MTS synthesized from the existing and the newly elicited properties ($p_1 - p_8$) and the new scenario specification sc'. This MTS still has maybe behaviour that can be used to prompt further elaboration, eventually leading to an MTS that covers the complete behaviour of the system up to the alphabet Act_{web}. In practice, it may not be necessary or even desirable to refine the MTS to a single LTS, and instead, certain aspects of behaviour may be left open to decisions further down the development process.

In summary, we have shown how to synthesize models from safety properties and scenarios by using the merge operation on MTSs. In addition, we have illustrated how a merged model that captures both scenarios and requirements may support behaviour model elaboration, and scenario and requirements elicitation.

VIII. Tool Support

We have developed a prototype implementation of a tool that supports construction and analysis of MTS models: the Modal Transition System Analyzer (MTSA) [9] (available at `http://www.doc.ic.ac.uk/~su2/MTSA.html`). MTS models are described in a textual language that includes traditional process algebra operators such as sequential and parallel composition, and hiding, in addition to the MTS merge operator. The tool also supports visualization

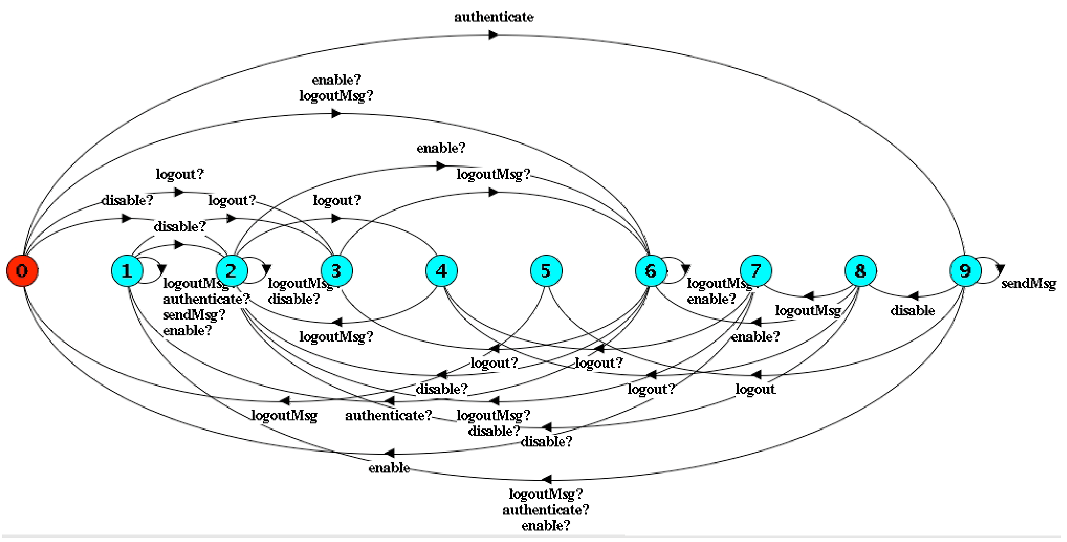


Fig. 12. MTS M_{both}: A merge of $M(P)$ in Figure 10 and $M(sc)$ in Figure 11.

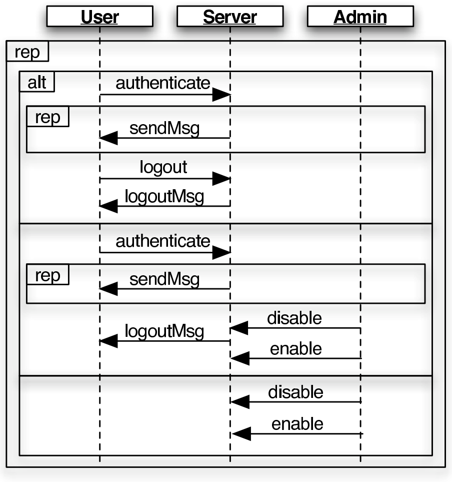


Fig. 14. Extended Webmail scenario specification (sc').

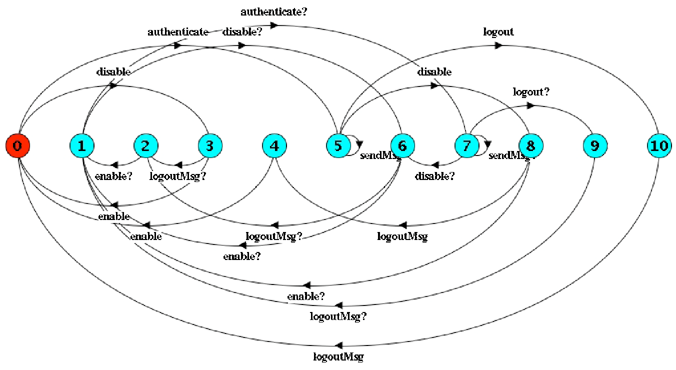


Fig. 15. Final model for Webmail: $M_P + M(p_4) + M(p_5 \wedge \ldots \wedge p_8) + M(sc')$.

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) [40] which supports specification and analysis of LTS models. MTSA extends the FSP process algebra to allow specification of MTSs. The compositional construction procedure that builds behaviour models from FSP has been adapted to construct MTS models, extending the semantics of existing FSP operators such as parallel composition and hiding. In addition, the graphical environment has been adapted and the visualization functionality extended to support depiction of MTSs.

MTSA implements 3-valued FLTL model checking of MTSs by reducing the problem to two classical FLTL model-checking runs on LTS models (see Theorem 4). Hence, MTSA builds on top of the model checking features of LTSA.

MTSA also provides a number of features which are specific to MTS models, such as checking MTS refinement and equivalence, consistency checking, merge of two MTSs, as well as the synthesis procedures described in this paper.

IX. Case Study: the Mine Pump

In this section, we discuss one of the case studies we have conducted: a pump controller system in a mine [28]. In this system, a pump controller is used to prevent the water in a mine sump from passing some threshold, and hence flooding the mine. To avoid the risk of explosion, the pump may only be on when there is no methane gas present in the mine. The pump controller monitors the water and methane levels by communicating with two sensors, and controls the pump in order to guarantee the safety properties of the pump system.

The case study presents a number of challenges when compared to the Webmail example used throughout the paper. First, the mine pump system requires a timed model in order to capture the urgency of actions such as switching the pump off to avoid an explosion when there is methane present. Consequently, properties must make use of an explicit *tick* event, signalling the successive ticks of a global

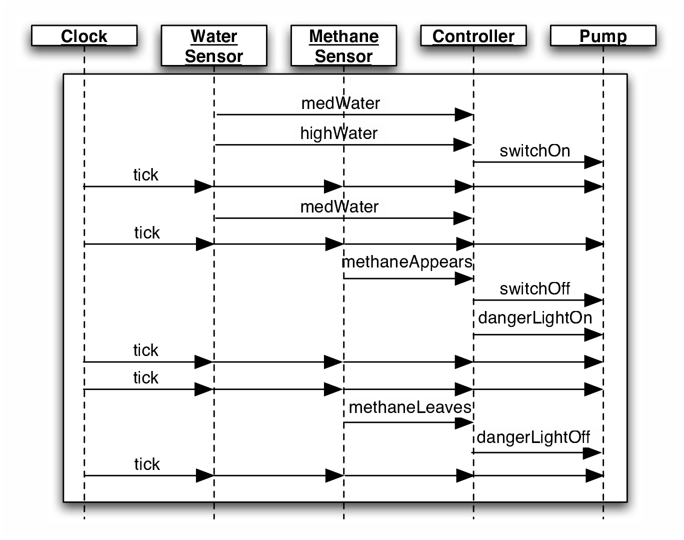


Fig. 16. A simple scenario for the mine pump system.

clock to which components with timed requirements synchronize. This corresponds to a standard approach to modeling discrete-time in event-based formalisms [44]. Second, Server is the only component with non-trivial behaviour in the Webmail example: there are no constraints on the behaviour of the User and the Admin. In contrast, the case study requires eliciting assumptions on the environment such as how the water and the methane level change. Finally, the initial requirements, described informally above, leave open a number of significant decisions in terms of the problem domain; these need to be made in order to elaborate the synthesized MTS.

The case study was conducted by iterating over a synthesize-analyze-elicit cycle: First, an MTS is synthesized from known properties and scenarios. Second, the maybe behaviour of the resulting MTS is analyzed to identify missing required and proscribed behaviour. The exploration of the maybe behaviour leads to the third stage, in which new scenarios and properties are elicited based on the user understanding of the problem domain. This process continues until a model with no maybe transitions is reached. In this case study, this occurred after the sixth iteration.

The synthesis phase of each iteration was conducted using the algorithms described in Sections V and VI and the MTS merge. The analysis phase was performed with the help of MTSA (see Section VIII): the maybe behaviour of the resulting MTS was inspected, both in its graphical and textual form, and then animated. To cope with the scale, abstraction and minimization features of MTSA were also used. Finally, given that the elicitation phase of the case study was conducted by the authors themselves rather than by analysts well familiar with the problem domain, we relied on the substantial amount of reference material that exists on the mine pump problem (e.g., [28]) in order to guide the elicitation process in each iteration.

We now describe the case study in more detail, presenting the initial properties that were used, some of the MTSs produced throughout the various iterations, and all

Act = {*switchOn*, *switchOff*, *methAppears*, *methLeaves*, *lowWater*, *medWater*, *highWater*, *switchDLOff*, *switchDLOn*, *tick*}

HighWater	=	⟨ *highWater*,{*lowWater*,*medWater*} ⟩ *initially False*
LowWater	=	⟨ *lowWater*,{*medWater*,*highWater*} ⟩ *initially True*
PumpOn	=	⟨ *switchOn*,*switchOff* ⟩ *initially False*
MethanePresent	=	⟨ *methAppears*, *methLeaves* ⟩ *initially False*

Fig. 17. Actions and fluents for the mine pump case study.

/* The pump shall be switched on if there is high water and no methane present */
$\phi_1 = \Box((\mathit{HighWater} \wedge \neg \mathit{MethanePresent}) \Rightarrow \mathbf{X}\ \mathit{PumpOn})$

/* The pump must be switched off if there is low water or there is methane present */
$\phi_2 = \Box((\mathit{LowWater} \vee \mathit{MethanePresent}) \Rightarrow \mathbf{X}\ \neg \mathit{PumpOn})$

/* If the pump is off, it should not be switched off */
$\phi_3 = \Box(\neg \mathit{PumpOn} \Rightarrow \mathbf{X}(\neg \mathit{switchOff}\ \mathbf{W}\ \mathit{PumpOn}))$

/* If the pump is on, it should not be switched on */
$\phi_4 = \Box(\mathit{PumpOn} \Rightarrow \mathbf{X}(\neg \mathit{switchOn}\ \mathbf{W}\ \neg \mathit{PumpOn}))$

Fig. 18. Initial properties of the mine pump case study.

of the properties and scenarios elicited. We also discuss the insights we gained of the problem domain and of the elaboration process.

A. The First Iteration

The case study starts with a scenario specification, σ, which includes scenarios such as the one depicted in Figure 16 and interpreted as follows:

> "After receiving messages from the water sensor indicating that the water level has gone over the medium and high water thresholds, the controller switches the pump on. As a consequence of the pumping, the water level goes down to medium. Later, the methane sensor informs the controller that there is a dangerous amount of methane in the mine. Hence, the controller switches the pump off to avoid an explosion and turns the danger light on to indicate that a dangerous level of methane is present in the mine. Once the methane sensor indicates that the level of methane has gone down, the danger light is switched off."

The system behaviour specified by σ can be depicted as an LTS shown in Figure 19, which has been synthesized from this set of scenarios, including the one in Figure 16.

The first iteration also includes four safety properties expressed in terms of the set of communicating actions of the mine pump controller (see *Act* in Figure 17) and four fluents (*HighWater*, *LowWater*, *PumpOn*, and *MethanePresent*, defined in Figure 17). The first two properties correspond to the key safety requirements of the mine pump system: prevent flooding (ϕ_1 in Figure 18) and prevent an explosion (ϕ_2 in Figure 18). The other two properties, ϕ_3 and ϕ_4, state that the pump must not be turned on (respectively, off) when it is already on (respectively, off).

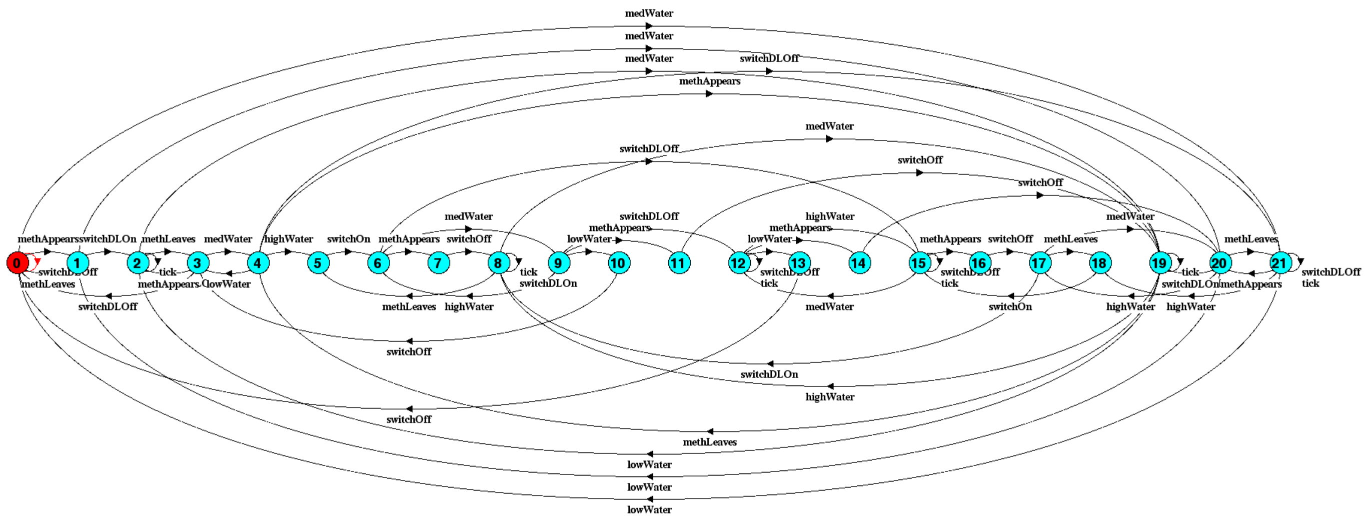


Fig. 19. LTS synthesized from the original scenario specification of the mine pump system ($L(\sigma)$).

The first iteration of the case study results in a partial operational model of the mine pump, M_1, which is the merge of the MTSs synthesized from the properties and the scenario specification: $M_1 = M(\phi_1) + M(\phi_2) + M(\phi_3) + M(\phi_4) + M(\sigma)$. Due to the size of M_1 (34 states and over 100 transitions), we present its textual rather than graphical representation for clarity (see Figure 20).

Note that M_1 predominantly contains maybe behaviour due to the fact that the scenario specification does not constrain behaviour (it only introduces required behaviour), and properties ϕ_1 to ϕ_4 form a rather loose specification of the intended system. Refinements of such a system result from the analysis of the maybe behaviour and elicitation of further properties and scenarios.

Although the textual representation of M_1 is more compact than its graphical counterpart, both remain unsatisfactory artifacts for supporting validation and analysis, due to their size. Going through the various states and transitions by hand even in the model this small leads to numerous confusions. This is why an automated tool, such as MTSA, that provides multiple techniques for supporting analysis is fundamental. To analyze M_1, we first animated the model using MTSA (see Figure 21), walking through a number of traces and inspecting the portions of the MTS that were reached by these traces.

Analysis of M_1 resulted in several findings. First, we identified a number of traces that exhibited undesired behaviour of the environment-controlled actions. For instance, M_1 exhibits the trace

$$\begin{aligned} tr_1 \quad = \quad & \mathit{highWater?}, \mathit{switchOn}, \mathit{lowWater?}, \mathit{switchOff?}, \\ & \mathit{highWater?}, \ldots \end{aligned}$$

which has water levels changing from *HighWater* to *LowWater* and back without going through medium water.

Given that an assumption for this system is that the water sensor of the mine pump does not fail to detect changes in the water level, and that water level cannot go from a low to a high level without going through a medium level, such traces should be prevented by the environment. Figure 22(a) depicts a transition system M_{water} describing the expected behaviour of water levels. Composing M_1 in parallel with M_{water}, prevents tr_1 from happening. Note that here we use parallel composition rather than merge, as we are composing models of different components (the controller and its environment) rather than two models of the same component.

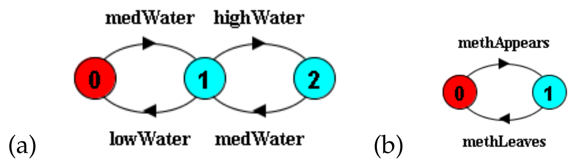


Fig. 22. (a) An MTS describing the water level environment; (b) An MTS describing the methane environment.

Another trace which prompted further elicitation of the environment behaviour models is

$$tr_2 \quad = \quad \mathit{methAppears}, \mathit{methAppears?}, \mathit{methAppears?}, \ldots$$

The resulting environment model, $M_{methane}$, shown in Figure 22(b), describes the assumption that the methane sensor signals readings to the controller only in those cases when the methane levels go above or below a certain threshold.

Further analysis and validation of M_1 prompted elicitation of preconditions for switching the danger light on and off, formalized as properties ϕ_5 and ϕ_6 in Figure 23. The danger light can be turned on to reflect that there is a dangerous level of methane present in the mine only if *MethanePresent* is true (ϕ_5). Analogously, the danger light can be turned off to reflect that the level of methane is not dangerous only if *MethanePresent* is false (ϕ_6). Note that these properties, together with the domain assumptions modelled in $M_{methane}$, guarantee that the danger light is not turned on (respectively, off) when it is already on (respectively, off).

B. The Remaining Iterations

$M_{methane}$ and M_{water} are the only two models elicited for the environment of the mine pump controller. In the

```
M1 = Q0,
Q0 =  ({lowWater?, methLeaves?, switchDLOn?, switchOff?} -> Q15
      |{switchDLOff, tick} -> Q19
      |medWater -> Q20
      |methAppears -> Q21
      |highWater? -> Q24
      |switchOn? -> Q28),
Q1 =  (switchDLOff -> Q5
      |medWater -> Q18
      |{highWater?, methLeaves?, switchDLOn?, tick?} -> Q27
      |lowWater? -> Q28
      |methAppears -> Q31),
Q2 =  (switchOn -> Q1),
Q3 =  ({switchDLOn, tick} -> Q3
      |lowWater -> Q14
      |{medWater?, methAppears?, switchDLOff?} -> Q16
      |methLeaves -> Q25
      |highWater -> Q33),
Q4 =  (switchOff -> Q3),
Q5 =  ({switchDLOff, tick} -> Q5
      |medWater -> Q10
      |methAppears -> Q26
      |{highWater?, methLeaves?, switchDLOn?} -> Q27
      |lowWater? -> Q28),
Q6 =  ({medWater?, methLeaves?, switchDLOff?, switchDLOn?, tick?} -> Q6
      |switchOn? -> Q8
      |lowWater? -> Q15
      |methAppears? -> Q16
      |highWater? -> Q24),
Q7 =  (switchOff -> Q23),
Q8 =  (switchOff? -> Q6
      |{medWater?, methLeaves?, switchDLOff?, switchDLOn?, tick?} -> Q8
      |highWater? -> Q27
      |lowWater? -> Q28
      |methAppears? -> Q29),
Q9 =  (switchOff -> Q19),
Q10 = (highWater -> Q5
      |switchOff? -> Q6
      |methAppears -> Q7
      |{medWater?, methLeaves?, switchDLOn?} -> Q8
      |lowWater -> Q9
      |{switchDLOff, tick} -> Q10),
Q11 = ({highWater?, methAppears?, switchDLOff?, switchDLOn?, tick?} -> Q11
      |lowWater? -> Q12
      |medWater? -> Q16
      |methLeaves? -> Q24),
Q12 = (highWater? -> Q11
      |{lowWater?, methAppears?, switchDLOff?, switchDLOn?, tick?} -> Q12
      |methLeaves? -> Q15
      |medWater? -> Q16),
Q13 = (methAppears -> Q14
      |{lowWater?, methLeaves?, switchDLOn?, tick?} -> Q15
      |switchDLOff -> Q19
      |highWater? -> Q24
      |medWater -> Q25),
Q14 = (medWater -> Q3
      |highWater? -> Q11
      |{lowWater?, methAppears?, switchDLOff?} -> Q12
      |methLeaves -> Q13
      |{switchDLOn, tick} -> Q14),
Q15 = (medWater? -> Q6
      |methAppears? -> Q12
      |{lowWater?, methLeaves?, switchDLOff?, switchDLOn?, tick?} -> Q15
      |highWater? -> Q24),
Q16 = (methLeaves? -> Q6
      |highWater? -> Q11
      |lowWater? -> Q12
      |{medWater?, methAppears?, switchDLOff?, switchDLOn?, tick?} -> Q16),
Q17 = (switchOff -> Q13),
Q18 = (highWater -> Q1
      |methAppears -> Q4
      |switchOff? -> Q6
      |{medWater?, methLeaves?, switchDLOn?, tick?} -> Q8
      |switchDLOff -> Q10
      |lowWater -> Q17),
Q19 = ({lowWater?, methLeaves?, switchDLOn?} -> Q15
      |{switchDLOff, tick} -> Q19
      |medWater -> Q20
      |methAppears -> Q21
      |highWater? -> Q24),
Q20 = ({medWater?, methLeaves?, switchDLOn?} -> Q6
      |switchOn? -> Q8
      |lowWater -> Q19
      |{switchDLOff, tick} -> Q20
      |methAppears -> Q23
      |highWater -> Q32),
Q21 = (highWater? -> Q11
      |{lowWater?, methAppears?, switchDLOff?, tick?} -> Q12
      |switchDLOn -> Q14
      |methLeaves -> Q19
      |medWater -> Q23),
Q22 = ({highWater?, methAppears?, switchDLOff?, tick?} -> Q11
      |lowWater? -> Q12
      |medWater -> Q23
      |methLeaves -> Q32
      |switchDLOn -> Q33),
Q23 = (switchDLOn -> Q3
      |{medWater?, methAppears?, switchDLOff?, tick?} -> Q16
      |methLeaves -> Q20
      |lowWater -> Q21
      |highWater -> Q22),
Q24 = (switchOn? -> Q27),
Q25 = (highWater -> Q2
      |methAppears -> Q3
      |{medWater?, methLeaves?, switchDLOn?, tick?} -> Q6
      |switchOn? -> Q8
      |lowWater -> Q13
      |switchDLOff -> Q20),
Q26 = (switchOff -> Q22),
Q27 = (medWater? -> Q8
      |{highWater?, methLeaves?, switchDLOff?, switchDLOn?, tick?} -> Q27
      |lowWater? -> Q28
      |methAppears? -> Q30),
Q28 = (switchOff? -> Q15),
Q29 = (switchOff? -> Q16),
Q30 = (switchOff? -> Q11),
Q31 = (switchOff -> Q33),
Q32 = (switchOn -> Q5),
Q33 = (methLeaves -> Q2
      |medWater -> Q3
      |{highWater?, methAppears?, switchDLOff?} -> Q11
      |lowWater? -> Q12
      |{switchDLOn, tick} -> Q33).
```

Fig. 20. Textual representation of the MTS synthesized from the initial scenario specification and properties of the mine pump system (M_1).

/* Methane present is a precondition for switchDLOn */
$\phi_5 =$
$\Box(\neg \mathit{MethanePresent} \Rightarrow (\neg \mathit{switchDLOn}\ \mathbf{W}\ \mathit{MethanePresent}))$

/* Methane not present is a precondition for switchDLOff */
$\phi_6 =$
$\Box(\mathit{MethanePresent} \Rightarrow (\neg \mathit{switchDLOff}\ \mathbf{W}\ \neg \mathit{MethanePresent}))$

Fig. 23. Properties elicited during the first iteration.

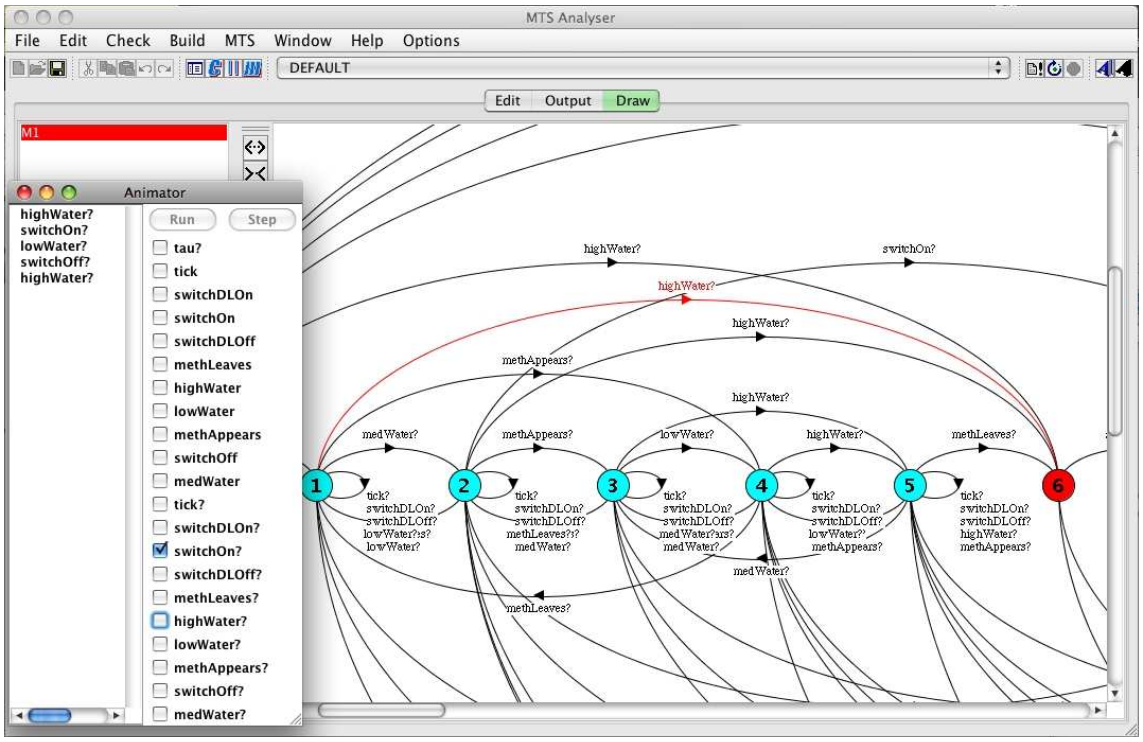
remainder of the case study, we performed the analysis and validation of successive refinements of the mine pump controller ($M_1, \ldots, M_n$) in the context of parallel composition with $M_{methane}$ and M_{water}, i.e., $M_i||M_{water}||M_{methane}$. However, to simplify the presentation, we shall simply refer to M_i rather than $M_i||M_{water}||M_{methane}$.

The second iteration of the case study starts with the synthesis of $M_2 = M_1 + M(\phi_5) + M(\phi_6)$, and the analysis of the resulting model prompts elicitation of two additional properties.

Event *switchOff* is enabled in M_2 through a maybe transition, even though the property ϕ_3 requires *switchOff* not to happen if *PumpOn* is false in the previous state. However, this property makes no restrictions on the first state of any execution (there is no previous state in which a pump could be on); thus, we add a property ϕ_7. Although this is an artefact of the semantics of FLTL which could have been avoided had we used an alternative semantics or introduced a *beginning of time* event, the example does show that the proposed approach can help identify the impact that the (inevitable) subtleties of formal property specification languages have on the system model.

A useful analysis of complex MTSs is to construct and validate M^+ and M^-, which can sometimes be significantly smaller than M and hence simpler to validate. Analyzing M^+ is particularly useful for identifying missing preconditions for actions, as all maybe transitions are converted to required transitions in such models. For instance, M_2^+ has 13 states against 34 states of M_2, and its analysis leads to strengthening the precondition for *switchOn*: It is not sufficient that the pump be off as required by ϕ_4; there should be water in the mine sump. Property ϕ_8 models this requirement by asserting that if *LowWater* is true, *switchOn* does not occur until *LowWater* becomes false.

The third iteration is based on the analysis of $M_3 = M_2 +$



highWater?
switchOn?
lowWater?
switchOff?
highWater?

Fig. 21. Animation of M_1.

/* Initially, the pump may not be switchedOff */
$\phi_7 = \neg \mathit{switchOff}$

/* Water not low is a precondition for switchOn */
$\phi_8 = \Box(\mathit{LowWater} \Rightarrow (\neg \mathit{switchOn}\ \mathbf{W}\ \neg \mathit{LowWater}))$

Fig. 24. Properties elicited during the second iteration.

$M(\phi_7) + M(\phi_8)$. A maybe trace that M_3 exhibits is

$$tr_3 = \mathit{medWater}, \mathit{highWater}, \mathit{switchOn}, \mathit{medWater}, \mathit{switchOff?}, \ldots$$

This trace leads to the question of exactly when the pump should be turned off. There are a number of different strategies that can be considered at this point. The first is to make the pump "eager", i.e., if there is any water to pump out, then the pump should remain on. A second strategy is a "lazy" pump: as long as there is no high water, the pump should be off. A third is a pump that minimizes its state changes: if it is on, it continues to be on until the water is low, and if it is off, it continues to be off until water is high. The loose initial specification for the controller allows other strategies as well.

We experimented with several of these strategies by exemplifying them in a scenario description, merging them with M_3 and analyzing the resulting behaviour. Automated synthesis and merging, supported by MTSA, significantly aided us with this exploration. For instance, when synthesizing some of the proposed strategies resulted in MTSs that were inconsistent with M_3, this was automatically flagged by the tool. Other strategies were perfectly consistent, could guarantee the high-level goals described at the beginning of

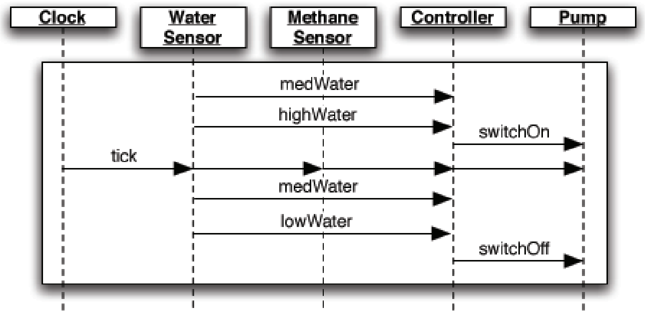


Fig. 25. An "eager" pump scenario, σ_1.

this section, and would have allowed us to reach LTS models for the mine pump controller that are valid alternatives to the one presented here. In this paper, we report on the choice to create an "eager" pump. Hence, we merged M_3 with another scenario, σ_1, described in Figure 25, resulting in a model $M_4 = M_3 + M(\sigma_1)$.

Analysis of M_4 led to a declarative description of when to switch the pump off that is consistent with σ_1: We found the need to strengthen the precondition of *switchOff*, as encoded in ϕ_9.

In addition, we elicited a complex property for the danger light resulting from the analysis of maybe traces such as

$$tr_4 = \mathit{medWater}, \mathit{highWater}, \mathit{switchOn}, \mathit{methaneAppears}, \mathit{switchOff}, \mathit{tick?}, \ldots$$

The trace shows that the controller is not reacting sufficiently fast to keep the state of the danger light appropri-

/* The pump must remain open as long as there is water to be pumped and no methane is present */
$\phi_9 = \Box(\mathbf{X}\ \mathit{switchOff} \Rightarrow (\mathit{LowWater} \vee \mathit{MethanePresent}))$

/* At each tick, the danger light shall be on if methane is present */
$\phi_{10} = \Box(\mathit{tick} \Rightarrow (\mathit{MethanePresent} \Leftrightarrow \mathit{DangerLightOn}))$

Fig. 26. Properties elicited during the third iteration.

ately set to reflect the levels of methane in the pump. A simplistic way to deal with this trace is to require that the light to be set immediately after methane appears:

$$\phi_{wrong} = \Box(\mathit{methaneAppears} \Rightarrow \mathbf{X}\ \mathit{switchDLOn})$$

This property is inconsistent with M_4. During the computation of $M_4 + M(\phi_{wrong})$, MTSA reports presence of a deadlock state reachable through a required trace. This means that there are no deadlock-free implementations of such a merge, or, equivalently, that construction of $(M_4 + M(\phi_{wrong}))^-$ results in an empty MTS.

MTSA can also be used to pinpoint the *cause* for the inconsistency between M_4 and ϕ_{wrong}. In principle, ϕ_{wrong} may not be inconsistent with any *individual* property (ϕ_1 to ϕ_9) or scenario (σ) previously elicited but with an emergent behaviour of a *subset* of these instead. The offending subset can be found either by starting from an empty subset and adding properties and scenarios into consideration, doing the merge and checking for inconsistency with property of interest, or by removing arbitrary properties or scenarios from the set of elicited artifacts, re-merging the remaining ones, and checking for inconsistency. In our case, it is property ϕ_2, which requires the pump to be immediately switched off when methane appears, that is inconsistent with ϕ_{wrong}: $(\phi_2 + \phi_{wrong})^-$ is the empty MTS. We discuss inconsistency identification and resolution further in Section X.

A natural reaction to this inconsistency is to change the "next" operator $\mathbf{X}$ into a "future" operator $\mathbf{F}$ in ϕ_{wrong}:

$$\phi'_{wrong} = \Box(\mathit{methaneAppears} \Rightarrow \mathbf{F}\ \mathit{switchDLOn})$$

This property is too weak and does not describe the urgency of keeping the danger light as aligned as possible with the level of sensed methane.

A possible refinement is to require that if methane is present and the danger light is off, then the light must be switched on before the next tick. However, if an analogous property for switching the danger light on is required, a contradiction is reached: if methane appears and disappears within one time unit what should the status be at the end of this time unit? This was a mistake that we made ourselves. We formalized the two properties for switching the light on and off and merged them, using MTSA, with the remainder of the properties and scenarios, resulting in an empty MTS. As a result, we were forced to backtrack and define a weaker property: the danger light should accurately reflect the state of methane in the mine at the end of each time unit, formalized as the property ϕ_{10}.

The fifth iteration, over the model $M_5 = M_4 + M(\sigma) + M(\sigma_1) + M(\phi_9) + M(\phi_{10})$, exhibits the trace

$$tr_5 = \mathit{medWater}, \mathit{switchOn?}, \ldots$$

which shows that the eager policy is not being enforced fully (ϕ_9 prevents the pump from being switched off but does not force the pump to become on). Hence, ϕ_{11} is added to force urgency over turning the pump on when the water is not low. However, this property contradicts the original scenario description σ (see Figure 19). This contradiction is identified when constructing the model $M_5 + M(\phi_{11})$ and observing that it deadlocks. Iteratively removing the previously elicited properties that comprise M_5, the contradiction between ϕ_{11} and the scenario specification σ is pinpointed, since the MTS $M(\phi_{11}) + M(\sigma)$ is deadlocking as well. Finally, an example of the contradiction can be produced by checking ϕ_{11} against $M(\sigma)$. In this case, the MTSA model checker returns a trace *medWater*, *tick*, which is required by the scenario specification σ while violating the property ϕ_{11}.

/* Pump must be on at tick if water is not low and methane not present */
$\phi_{11} =$
$\Box(\mathit{tick} \Rightarrow ((\neg \mathit{LowWater} \wedge \neg \mathit{MethanePresent}) \Rightarrow \mathit{PumpOn}))$

/* Danger light off is a precondition for switching the danger light on */
$\phi_{12} =$
$\Box(\mathit{DangerLightOn} \Rightarrow \mathbf{X}(\neg \mathit{switchDLOn}\ \mathbf{W}\ \neg \mathit{DangerLightOn}))$

/* Danger light on is a precondition of switching the danger light off */
$\phi_{13} =$
$\Box(\neg \mathit{DangerLightOn} \Rightarrow \mathbf{X}(\neg \mathit{switchDLOff}\ \mathbf{W}\ \mathit{DangerLightOn}))$

Fig. 27. Properties elicited during the fifth iteration.

Summarizing, the fifth iteration involved modifying the scenario specification σ by *removing* the traces that violate ϕ_{11}. We call the resulting scenario description σ'.

The remaining two properties elicited in this iteration are that the danger light cannot be switched on when it is already on (ϕ_{12}), nor off while it is off (ϕ_{13}).

The final iteration involves analyzing

$$M_6 = M(\phi_1) + \cdots + M(\phi_{13}) + M(\sigma') + M(\sigma_1).$$

A careful analysis of M_6 revealed that all remaining maybe transitions should be converted to required transitions. This led to a fully refined MTS and the conclusion of the case study. We now explain why the decision to convert remaining maybe transitions into required ones is in order.

The maybe transitions on actions controlled by the environment (i.e., *methLeaves*, *tick*, *methAppears*, *lowWater*, *medWater*, and *highWater*) should be converted to required transitions to ensure that the controller does not constrain the behaviour of the environment unnecessarily. The maybe transitions on actions controlled by the pump controller (i.e., *switchOn*, *switchOff*, *switchDLOn*, and *switchDLOff*) are either transitions whose initial state has one outgoing transition, or states in which *tick* is not enabled but other environment actions are. The former indicates that the transition must be required; otherwise, the implementation

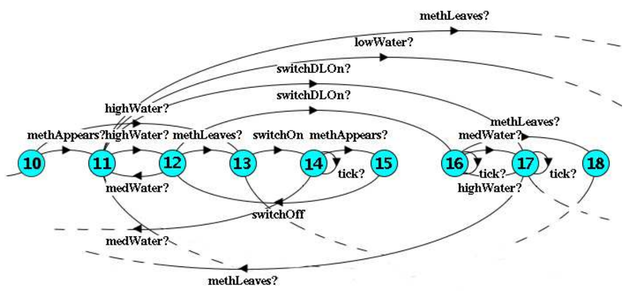


Fig. 28. A fragment of M_6.

would deadlock. The latter indicates that there is some urgency requirement that remains to be fulfilled. Consider properties ϕ_{10} and ϕ_{11} which restrict the occurrence of the *tick* event. These properties deliberately leave underspecified when, within a time unit, the requirement must be satisfied. Now consider the fragment of M_6 depicted in Figure 28 where the leftmost state is the one in which there is no methane present, water level is medium and the pump is off. Once methane appears (transition between states 10 and 11), time will not go forward until either the danger light is switched on or methane leaves again (and hence, there is no urgency for switching the light on). There can be a number of ways to implement the maybe transitions on *switchDLOn* from states 1 and 2: one, both, or neither of these transitions can be implemented. However, the sensible choice is to require both transitions because removing one of them would introduce unnecessary assumptions on the environment (e.g., removing a transition from state 1 to 6 introduces a requirement that the environment will exhibit *lowWater*, *methLeaves*, or *highWater* before the next *tick* event).

In summary, the maybe behaviour of M_6 is converted to the required behaviour, resulting in the LTS in Figure 29, which is a reasonable choice for the implementation model of the pump controller.

C. Summary

In this section, we have reported on the use of scenario and property synthesis in conjunction with analysis of MTS models using MTSA to support model elaboration. We described just one of the many elaborations which could have been done as the result of analyzing the unknown behaviour. The story we presented was somewhat simplified. In reality, we made numerous incorrect decisions in our understanding of the domain, in the formalization of properties, and in the exemplification of the desired behaviour. We have reported on some of these to show how our approach supports exploring and validating decisions regarding the system behaviour, how incorrect decisions can lead to inconsistencies later on in the elaboration process, and how these can be traced back to their source.

The model resulting from the first iteration is less refined than models that have been previously constructed from the analogous properties by hand (e.g., [3]) and automatically (e.g., [36]), which indicates that hidden assumptions were made at the modelling or synthesis time in both approaches. For instance, in [36], the mine pump was modeled using LTSs, and thus the synthesis algorithm had to pick one of the possible implementations for the initial set of requirements. The LTS synthesized by this approach is a "greedy" implementation that attempts to include as many behaviours as possible without violating any properties. Although the resulting model satisfies the properties known initially, it does not promote (or allow) eliciting further properties. For example, identification of different strategies for keeping the pump on or off at intermediate levels of water was one important outcome of avoiding arbitrary decisions during the model synthesis.

The iterative refinement of the original behaviour description provided by the properties ϕ_1 to ϕ_4 and the scenario specification σ, embodied in the MTS M_1, is sound. As proven in the previous sections, i) the MTS refinement narrows down the set of implementations; hence, as new properties and scenarios are elicited and merged, the original properties and scenarios continue to hold (unless inconsistency has been found and some of the properties or scenarios had to be removed); ii) the MTS merge characterizes logical conjunction; hence, as the elicitation process continues, the resulting MTS precisely captures all deadlock-free implementations that satisfy the previously elicited as well as the new properties.

It is worth mentioning that the elicited FLTL properties coincide with well-known temporal patterns: urgency, invariance, precondition, etc. These or similar patterns have been identified in goal-oriented requirements engineering approaches such as KAOS [52] and property patterns such as the Dwyer system [10]. We therefore expect that our approach can benefit from the use of patterns as well, making it more accessible and decreasing the need of formality for behaviour model elicitation.

We used a number of analysis techniques to support eliciting properties and scenarios that would refine maybe behaviour. We performed animations of the MTS models using the MTSA tool, exploiting their operational nature. We did not use graphical animation toolkits such as the one described in [41], because these have been designed for traditional behaviour models such as LTSs. However, these approaches can be adapted straightforwardly if some visual convention is used to distinguish between maybe and required behaviour.

We also relied on inspection of synthesized MTSs, both in their textual and graphical forms, as produced by the MTSA tool. For larger models, validation of minimized M^+ and M^- models was very helpful. A useful technique that could be adapted to aid inspection is the conversion of MTSs into some hierarchical representation, like Statecharts [15], and visualization of *abstractions* of an MTS using action hiding and minimization [40], [8]. The latter, although defined and implemented for strong refinement [8], has yet to be implemented for our case which requires weak refinement.

We used model-checking intensively to support the analysis. The introduction of an inconsistent property in an iteration of the synthesize-analyze-elicit cycle is detected using a deadlock freedom check. This amounts to checking that a given MTS has at least one deadlock-free implementation, which is done by MTSA. Having identified an inconsistency,

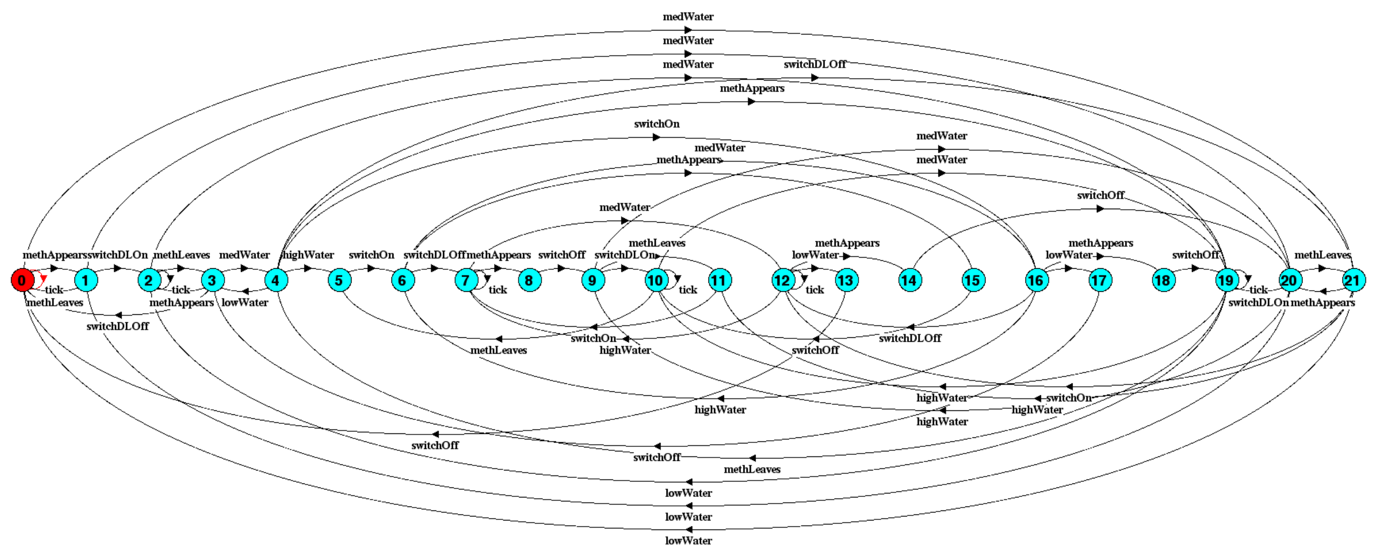


Fig. 29. The final behavioral model of the mine pump controller.

examples to aid comprehension can be produced by model checking the elicited property against the system model. MTSA produces a counter-example that may help identify the source of the problem: either the elicited property is incorrect, or it has been incorrectly formalized, or some of the previously elicited properties or scenarios are incorrect or incorrectly formalized. In the latter case, checking the elicited property against a smaller subset of the previously elicited properties or scenarios would help pinpoint the source of inconsistency.

X. Discussion and Related Work

In this section, we discuss the results presented in this paper and some of the decisions we have made, comparing our approach to related work.

A. On Safety Properties

In this paper, we limit our analysis to safety properties. This restriction is reasonable from a requirements perspective (it is standard in such approaches as goal-oriented requirements engineering [51]): instead of handling liveness properties such as $\Diamond\phi$, we assume that if the system is required to do something eventually, surely there is a bound on the acceptable time in which this must occur. The bounded response results in a safety property. Hence, approaches such as [51] adopt bounded temporal operators, such as in $\Diamond_{\leq q}\phi$, which means "ϕ holds eventually but in less than q time units" to provide a richer syntax for describing safety properties. Bounded temporal operators can be easily expressed in FLTL.

From a practical perspective, the FLTL to MTS synthesis algorithm identifies properties that are not safety automatically, as these result in intermediate Büchi automata which do not have a unique accepting trap state.

Should we choose to support synthesis from liveness properties directly, we would require construction of models with non-trivial accepting criteria for infinite traces – such as those provided by Büchi automata.

B. MTSs over Infinite Traces

In Linear Temporal Logic (LTL) [42] and thus in FLTL, formulae are evaluated over infinite traces. Consequently, checking satisfaction of an LTL formula over an operational model that has states with no outgoing transitions requires a work-around: either finite traces are extended with (implicit or explicit) self-loops [19], or LTL semantics is extended to enable explicit reasoning about such traces [17]. When defining FLTL, we found that giving it semantics w.r.t. finite traces lacks elegance and loses the intuitive meaning behind many FLTL formulae. Instead, we chose to restrict MTSs (and therefore LTSs) to those that do not have finite traces altogether. This decision not only produces an elegant formalism but also resolves an expressiveness issue with MTSs, which we discuss below.

Consider an FLTL formula $a \vee b$, and note the implicit partiality: it is not known whether a or b should occur, as long as one (or both) do. The MTS $\mathcal{A}$ in Figure 5 is intended to capture the implementations that satisfy this formula, where transitions on a and b are modelled with maybe behaviour from the initial state.

Unfortunately, there is no way to specify within the MTS formalism that a transition on either label can be refined to false in the implementation, as long as at least one other transition is true. Specifically, a possible refinement of $\mathcal{A}$ is the empty LTS, i.e., a finite-trace model. Whether we extend this trace to a self-loop or ignore the finite-trace behaviour, this model does not preserve $a \vee b$. By considering just infinite-trace MTSs, we restrict the space of allowable refinements, avoiding such "bad" implementations. The infinite-state requirement ensures that a transition on either a or b is required in the initial state of any implementation of $\mathcal{A}$ (e.g., model $\mathcal{B}$ in Figure 5).

An alternative solution to the problem of synthesis of properties with disjunctions is to build disjunctive modal transition systems (DMTS) [32]. States in DMTSs have sets of outgoing possible transitions where at least one transition of each set must be provided in an implementation of the

DMTS. The cost of this solution is a more complex modeling formalism that may encumber effective specification and understanding of behaviour models.

C. 3-valued FLTL

In Section IV, we argued that our 3-valued FLTL has thorough semantics. In contrast, conventional model-checking approaches implement *compositional* semantics – computing the value of the formula from the values of its subformulas. Compositional semantics is typically less precise than thorough: the *maybe* value returned by the model-checkers may not correspond to the existence of implementations in which the property holds and those where it fails. However, for a logic with just universal quantifiers, such as FLTL, compositional and thorough semantics coincide [14], enabling Theorem 4 and thus the analysis with the complexity of compositional model-checking.

The relation between MTSs and 3-valued modal μ-calculus logics [3], [23] has been studied, and these logics have been shown to be a good fit for characterizing the notions of refinement, observational refinement, merging and consistency. We have chosen a less expressive logic, namely, FLTL, that is preserved by refinement but does not characterize these notions. Our choice of a linear temporal logic is in line with other requirements engineering approaches [13], [52], [25] and property specification languages [10].

The motivation for choosing a fluent-based logic is that it provides a uniform framework for specifying and model-checking state-based temporal properties in event-based transition systems [13]. The fluent definitions are therefore useful for automatic synthesis of event-based operational models from state-based declarative properties.

The 3-valued FLTL logic presented in this paper enables specification and model-checking of both safety and liveness properties. Therefore, while Section V restricts the synthesis of MTSs to safety properties, implementations obtained via refinement will preserve all properties (including liveness) that the synthesized MTSs may have.

D. Alphabet Extension

In this paper, we have so far ignored the issue of alphabet extension, assuming instead that all properties are defined over the same alphabet *Act*. In practice, fixing *Act* is not practical, as the process of elaboration involves discovery of new relevant actions. Hence, elaboration should support augmenting the universe of known actions with new ones. The results we present in this paper can be extended to deal with alphabet extensions, relying mainly on the notion of *observational* refinement and properties of merge with respect to it [48], [3]. Specifically, our previous study of merge handles different alphabets, as reported in [48], [3].

If *Act* is not fixed, we cannot guarantee that the result of the merge is always unique. In other words, there may not always be a unique minimal common refinement of the models being merged. This can even occur for *non-deterministic* models with the same alphabet [21], but does not apply in this paper as we synthesize and merge deterministic models.

Note that Theorem 5 in Section V which relates LTS composition with logical conjunction, requires formulae to be closed under stuttering [1] if each formula being used for synthesis is allowed a different universe *Act* [37]. However, this restriction is not needed for MTSs: Theorem 9, which relates merge with conjunction, holds in all cases when the minimal common observational refinement is unique.

E. Elaboration Process

The synthesis techniques presented in this paper are not intended to be used in a one-off fashion. Rather, synthesis is expected to be used iteratively, adding, va merge, more information to the partial behaviour model being constructed. This elaboration process consisting of synthesis, merge, analysis and elicitation, exemplified in the case study section, is guaranteed to be monotonic with respect to the MTS refinement. In other words, as new properties and scenarios that are consistent with MTS under analysis are elicited, merging-in the new properties and scenarios preserves the required and proscribed behaviour specified up to that point. If, however, an inconsistency is reached, the source of inconsistency can be traced by decomposing the MTS under analysis into the various models that represent the scenarios and properties elicited previously.

We envisage that the result of the elaboration process includes not only the properties and scenarios elicited throughout, but also the MTS that characterizes these scenarios and properties. If the modeller has confidence that all important scenarios and properties have been elicited, an arbitrary implementation of the resulting MTS can be chosen (easy choices are M^+ and M^-). Such implementation can be used to move towards a system design and implementation supported by the variety of techniques based on traditional behaviour models such as LTSs.

F. Related Work

A number of approaches to building event-based models from properties exist [30], [50], [43], [25], [37], [38]. For instance, [30] proposed a technique for automatically translating a goal-oriented requirements model into a tabular event-based specification in the form of SCR [18]. [50], [43] developed behaviour model synthesis techniques to support animation and validation of property-based specifications. In [25], Formal Tropos goal models are translated into the event-based specification language Promela for verification using the SPIN tool. All of these approaches, as well as [37], build *one* of the many possible event-based models that satisfy the given properties. We addressed limitations of such approaches in Section II. An alternative, presented in [38], requires that the set of properties be strong enough to allow for a *unique*, up to bisimulation, operational model that satisfies them. Our work aims to support elaboration so as to potentially achieve such a strong set of properties.

Operational models have also been built from scenario descriptions (e.g. [49], [29], [27]). These approaches benefit from simple, intuitive notations that are widely used and well-suited for developing first approximations of the intended system behaviour. The operational nature of scenarios and the describe-by-example philosophy they embody are both an advantage, in terms of ease of use and adoption, and a disadvantage, in terms of having a generative semantics in which all behaviours must be explicitly described, and in terms of the number of scenarios that may be required to describe complex behaviours. We discussed limitations of such approaches previously.

The work by van Lamsweerde et al. [6] is related to ours in that it also considers scenarios and safety properties as an input to synthesis. A learning algorithm is used to synthesize an LTS model from examples of intended and proscribed system behaviour. The algorithm also provides feedback in terms of what-if questions in order to avoid over-generalization while learning. Safety properties are used to prune the number of what-if questions that are presented to the user. The difference with our work, however, is that the resulting LTS does not model the safety properties; it is simply constructed from scenarios that satisfy the safety properties. Hence, the LTS is a lower bound on the intended behaviour of the system and as such has the limitations discussed previously in the paper.

Live Sequence Charts (LSCs) [16] augment sequence charts with the goal of describing existential and universal behaviour. However, synthesis approaches for LSCs (e.g., [2], [45]) build traditional behaviour models and still do not support modeling and reasoning about possible yet not required system behavior. Extending LSC synthesis to characterize all models representable by an LSC specification would require a more expressive synthesis target (e.g., 3-valued Büchi automata) than the one used in this paper to allow to deal with liveness properties.

Modal Transition Systems [35] and their variants such as Disjunctive MTSs [32] and Mixed Transition Systems [7], have been studied from a theoretical standpoint for some time (e.g., [20], [22]) and are increasingly being proposed for modelling and reasoning about software systems from different perspectives such as program analysis [23], early requirements and design [48] and software product lines [12], [33]. These approaches assume the existence of a partial behaviour model and do not focus, as the work in this paper, on supporting the construction and elaboration of such models.

Finally, there have been approaches, e.g., [31], for solving the problem complementary to ours, i.e., *splitting* the merged model into individual models, to facilitate a reasoning task.

XI. Summary and Future Work

In this paper, we have presented an automated technique for constructing behaviour models from *both* safety properties and scenario-based specifications. We have argued that classical state machine models such as LTSs are insufficiently expressive to adequately support this procedure and presented synthesis algorithms that produce models in a more expressive formalism, namely, Modal Transition System. We have shown how synthesis of MTS models supports behaviour model elaboration in addition to requirements and scenario elicitation. Our approach integrates well with existing techniques such as goal- [5], [51] and scenario-based [46] requirements engineering.

There are a number of research issues that we aim to address in the future. We aim to provide automated support for inconsistency detection and resolution, and further integrate our work with existing work approaches to requirements engineering such as goal-oriented RE [51]. In addition, we plan to further investigate techniques for synthesis and merging of specifications that have different vocabularies. Finally, the tool support for the processes described in this paper requires additional work aimed to improve its usability and efficiency.

Acknowledgements. We thank anonymous ICSE and TSE referees for their comments first on a conference version of this paper [47] and then on an earlier version of the journal paper. We gratefully acknowledge the support of CONICET, ERC StG 204853-2, PICT 11-32440, UBACYT and NSERC. Nicolás D'Ippolito and Dario Fischbein contributed to the development of the MTSA tool.

References

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

[2] Y. Bontemps and P. Heymans. "From Live Sequence Charts to State Machines and Back: A Guided Tour". *IEEE Transactions on Software Engineering*, 31(12):999–1014, 2005.

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

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

[5] J. Castro, M. Kolp, and J. Mylopoulos. "Towards Requirements-Driven Information Systems Engineering: the Tropos Project". *Journal of Information Systems*, 27(6):365–389, 2002.

[6] C. Damas, B. Lambeau, and A. van Lamsweerde. "Scenarios, Goals, and State Machines: A Win-Win Partnership for Model Synthesis". In *Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'06)*, pages 197–207, 2006.

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

[8] N. D'Ippolito. "MTSA: A Model Checker for Modal Transition Systems". Master's thesis, University of Buenos Aires, Computing Department, December 2007.

[9] N. D'Ippolito, D. Fischbein, M. Chechik, and S. Uchitel. "MTSA: The Modal Transition System Analyzer". In *Proceedings of Tools Track of International Conference on Automated Software Engineering (ASE'08)*, September 2008.

[10] M. Dwyer, G. Avrunin, and J. Corbett. "Patterns in Property Specifications for Finite-State Verification". In *Proceedings of International Conference on Software Engineering (ICSE'99)*, pages 411–420, 1999.

[11] D. Fischbein and S. Uchitel. "On Correct and Complete Strong Merging of Partial Behaviour Models". In *Proceedings of ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'08)*, November 2008.

[12] D. Fischbein, S. Uchitel, and V. Braberman. "A Foundation for Behavioural Conformance in Software Product Line Architectures". In *Proceedings of ISSTA'06 Workshop on Role of Soft. Arch. for Testing (ROSATEA'06)*, pages 39–48, New York, NY, USA, 2006. ACM.

[13] D. Giannakopoulou and J. Magee. "Fluent Model Checking for Event-Based Systems". In *Proceedings of Joint European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'03)*, 2003.

[14] A. Gurfinkel and M. Chechik. "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, 2005.

[15] D. Harel. "StateCharts: A Visual Formalism for Complex Systems". *Science of Computer Programming*, 8:231–274, 1987.

[16] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.

[17] K. Havelund and G. Rosu. "Monitoring Programs Using Rewriting". In *Proceedings of International Conference on Automated Software Engineering (ASE'01)*, pages 135–143, 2001.

[18] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. "Automated Consistency Checking of Requirements Specifications". *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[19] G.J. Holzmann. "The Model Checker SPIN". *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

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

[21] A. Hussain and M. Huth. "Automata Games for Multiple-model Checking". *Electronic Notes in Theoretical Computer Science*, 115:401–421, 2006.

[22] M. Huth, R. Jagadeesan, and D. Schmidt. "A Domain Equation for Refinement of Partial Systems". *Mathematical Structures in Computer Science*, 14(4):469–505, 2004.

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

[24] ITU. Recommendation z.120: Message sequence charts. *ITU*, 2000.

[25] R. Kazhamiakin, M. Pistore, and M. Roveri. "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, 2004.

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

[27] K. Koskimies and E. MLkinen. "Automatic Synthesis of State Machines from Trace Diagrams". *Software Practice and Experience*, 24(7):643–658, 1994.

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

[29] I. Krüger, R. Grosu, P. Scholz, and M. Broy. "From MSCs to Statecharts". In *Distributed and Parallel Embedded Systems*. Kluwer Academic Publishers, 1999.

[30] R. De Landtsheer, E. Letier, and A. van Lamsweerde. "Deriving Tabular Event-Based Specifications from Goal-Oriented Requirements Models". In *Proceedings of IEEE International Symposium on Requirements Engineering (RE'03)*, pages 200–212, 2003.

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

[32] K. Larsen and L. Xinxin. "Equation Solving Using Modal Transition Systems". In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS'90)*, pages 108–117, July 1990.

[33] K. G. Larsen, U. Nyman, and A. Wasowski. "Modal I/O Automata for Interface and Product Line Theories". In *European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, 2007.

[34] K. G. Larsen, B. Steffen, and C. Weise. "A Constraint Oriented Proof Methodology based on Modal Transition Systems". In *Proceedings of 1st International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95)*, volume 1019 of *LNCS*, pages 17–40, 1995.

[35] K.G. Larsen and B. Thomsen. "A Modal Process Logic". In *Proceedings of IEEE Symposium on Logic in Computer Science (LICS'88)*, pages 203–210, 1988.

[36] E. Letier, J. Kramer, J. Magee, and S. Uchitel. "Fluent Temporal Logic for Discrete-time Event-Based Models". In *Proceedings of ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'05)*, pages 70–79, 2005.

[37] E. Letier, J. Kramer, J. Magee, and S. Uchitel. "Deriving Event-Based Transition Systems from Goal-Oriented Requirements Models". Technical Report 02/2006, Imperial College, 2006.

[38] E. Letier and A. van Lamsweerde. "Deriving Operational Software Specifications from System Goals". In *Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'02)*, pages 119–128, 2002.

[39] O. Lichtenstein and A. Pnueli. "Checking that Finite State Concurrent Programs Satisfy Their Linear Specification". In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages (POPL'85)*, pages 97–107. ACM, 1985.

[40] J. Magee and J. Kramer. *Concurrency - State Models and Java Programs*. John Wiley, 1999.

[41] J. Magee, N. Pryce, D. Giannakopoulou, and J. Kramer. "Graphical Animation of Behavior Models". In *Proceedings of International Conference on Software Engineering (ICSE'00)*, pages 499–508, 2000.

[42] Z. Manna and A. Pnueli. "Verification of Concurrent Programs: A Temporal Proof System". Technical report, Department of Computer Science, Stanford University, 1983.

[43] Ch. Ponsard, P. Massonet, A. Rifaut, J.F. Molderez, A. van Lamsweerde, and H. Tran Van. "Early Verification and Validation of Mission-Critical Systems". In *Proceedings of Formal Methods in Critical Systems (FMICS'04)*, 2004.

[44] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[45] J. Sun and J. Song Dong. "Design Synthesis from Interaction and State-Based Specifications". *IEEE Transactions on Software Engineering*, 32(6), 2006.

[46] A. G. Sutcliffe, N. A.M. Maiden, S. Minocha, and D. Manuel. "Supporting Scenario-Based Requirements Engineering". *IEEE Transactions on Software Engineering*, 24(12):1072–1088, 1998.

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

[48] S. Uchitel and M. Chechik. "Merging Partial Behavioural Models". In *Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'04)*, pages 43–52, 2004.

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

[50] H. Tran Van, A. van Lamsweerde, P. Massonet, and Ch. Ponsard. "Goal-Oriented Requirements Animation". In *Proceedings of IEEE International Symposium on Requirements Engineering (RE'04)*, pages 218–228, 2004.

[51] A. van Lamsweerde. "Tutorial: Goal-Oriented Requirements Engineering: From System Objectives to UML Models to Precise Software Specifications". In *Proceedings of International Conference on Software Engineering (ICSE'03)*, pages 744–745, 2003.

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

[53] M. Y. Vardi and P. Wolper. "An Automata-Theoretic Approach to Automatic Program Verification". In *Proceedings of 1st Symposium on Logic in Computer Science (LICS'86)*, pages 322–331, Cambridge MA, 1986.