Generating Unbiased Models from Temporal Properties

Greg Brunet, Sebastian Uchitel
Department of Computing
Imperial College
180 Queen's Gate
London, SW7 2RH, UK
[gwb,s.uchitel]@doc.ic.ac.uk

Marsha Chechik
Department of Computer Science
University of Toronto
40 St. George Street
Toronto, Canada M5S 2E4
chechik@cs.toronto.edu

ABSTRACT

Synthesis of event-based behaviour models from declarative statements on the required behaviour of systems can not only significantly reduce the effort of model construction but also provide a bridge between approaches geared towards requirements analysis and those geared towards reasoning about system design at the architecture level.

In this paper, we argue that partial behaviour models are a natural target for synthesis from a set of properties of intended system behaviour. These models are capable of distinguishing behaviours that are required and proscribed by properties from behaviours that may or may not be present in a final implementation. Specifically, we present a method to automatically construct Modal Transition Systems from safety properties expressed in Fluent Linear Temporal Logic. The resulting models characterize all Labelled Transition System implementations that satisfy the properties and consequently do not introduce implementation bias. The partial models can be used for validation and further elicitation of requirements, and iteratively refined into a desired implementation.

1. INTRODUCTION

Event-based behavioural models such as Labelled Transition Systems (LTSs) [14] are convenient formalisms for modelling and reasoning about system behaviour at the architectural level. They 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. 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. Behavioural 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 behavioural models have been investigated (e.g., [18, 29, 26, 13, 21, 17, 28]).

One approach is to automatically construct behavioural models from declarative statements, such as goal models or other properties describing the requirements of a system (e.g., [18, 26, 21, 13]). Properties prune the space of acceptable behaviours and can be composed with other properties, which reduces the space of possible implementations. This is a desirable alternative to operational (generative) approaches such as scenario-synthesis [28, 17] that must enumerate all possible behaviours.

Synthesis from declarative specifications has a number of advantages. Firstly, automatic synthesis delivers executable models early in the requirements process, enabling a wide range of validation techniques such as animations, simulations, and scenario-based techniques. Secondly, it provides a bridge between two modelling worlds: one well suited for requirements analysis, such as obstacle [31] and conflict analysis [30], the another that is well suited for architectural and high-level design analysis [23].

A current limitation of approaches to synthesis from properties is that the target model is assumed to be a complete description of system behaviour up to some level of abstraction, i.e., the state machine is assumed to completely describe the system behaviour with respect to a fixed alphabet of actions. Synthesis of behavioural models (e.g. LTSs) from properties necessarily involves making assumptions on how the system is to realize the required properties. As LTSs do not allow for under-specification within a given alphabet, the complete behaviour of the system with respect to the alphabet of the required properties must be fixed. This means that the synthesis procedure will be biased with respect to the choice of implementation. For instance, in [21], the synthesized model is required to perform all possible actions as long as they do not violate the given properties. However, it is more likely that the strategy for implementation will be to develop a system that does as little as possible while satisfying all requirements. Synthesizing biased models precludes incremental elaboration, exploration of the design space, and embeds assumptions in the synthesis that are not necessarily valid.

Since property-based descriptions are meant to specify only part of the intended system behavior, it is more appropriate to synthesize state-based models that explicitly model currently unknown aspects of the behaviour. Such models can distinguish between positive, negative and unknown behaviours. Positive behaviour refers to the behaviour the system is required to exhibit, negative behaviour refers to the behaviour the system is required *not* to exhibit, and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

unknown behaviour could become positive or negative, but the choice has not yet been made. Models that distinguish between these kinds of behaviour, such as Modal Transition Systems (MTSs) [19], are referred to as *partial behavioural models*, and can help solve the bias problem.

If the semantics of a partial behavioural model is thought of as the set of implementations it allows, then a natural interpretation of synthesis is to build a partial behavioural model that characterizes all possible implementations that satisfy a given set of properties, thereby enabling the choice of, or step-wise refinement towards, a desired implementation. For instance, the semantics of an MTS can be thought of in terms of the set of LTSs that can be refined from it. If an MTS that characterizes all possible LTSs that satisfy given properties is synthesized, the implementation can be reached through successive refinements of the MTS, possibly assisted by walk-throughs, animations, scenario-based elicitation, and composition with other operational models, either synthesized or developed in traditional ways (e.g., process calculi, graphical notations).

In this paper, we present a technique for automatically generating MTS models from safety properties expressed in Fluent Linear Temporal Logic (FLTL). Linear temporal logics are widely used to describe behaviour requirements [6, 31, 13]. The motivation for choosing FLTL is that it provides a uniform framework for specifying and model-checking state-based temporal properties to be satisfied by event-based transition systems [6]. Our choice of restricting the approach to safety properties does not limit the applicability of our technique, for reasons discussed in detail in Section 4. In addition to the main contribution of this paper, i.e., the synthesis algorithm itself, we define 3-valued FLTL over MTSs, prove that 3-valued FLTL is preserved by refinement, and that merging MTSs (a process defined in [27]) corresponds to logical conjunction of FLTL properties. We illustrate our results on a case study.

The rest of the paper is organized as follows. After presenting the necessary background in Section 2, we define and study a 3-valued version of FLTL in Section 3. Section 4 presents the synthesis algorithm and proves that merging is logical conjunction. In Section 5, we apply results of this paper, illustrating construction of a partial model and its elaboration into a desired implementation. We discuss our work and compare it to related approaches in Section 6, and give a summary and directions for future work in Section 7.

2. BACKGROUND

In this section, we review the notion of and operations over transition systems, and fix the notation.

DEFINITION 1. (Labelled Transition System) [14] *Assume States is a universal set of states, and Act is a universal set of observable action labels. An* LTS *is a tuple* $P = (S, L, \Delta, s_0)$*, where* $S \subseteq$ *States is a finite set of states,* $L \subseteq$ *Act is a set of labels,* $\Delta \subseteq (S \times L \times S)$ *is a transition relation, and* $s_0 \in S$ *is the initial state.*

Modal transition systems (MTSs) [19], which allow for explicit modelling of what is *not* known, extend LTSs with an additional set of transitions that model the interactions with the environment that the system cannot be guaranteed to provide, and equally cannot be guaranteed to prohibit.

DEFINITION 2. (Modal Transition System) *An* MTS M *is a structure* $(S, L, \Delta^r, \Delta^p, s_0)$*, where* $\Delta^r \subseteq \Delta^p$*,* (S, L, Δ^r, s_0) *is an LTS representing* required *transitions of the system and* (S, L, Δ^p, s_0) *is an LTS representing* possible *(but not necessarily required) transitions.*

We refer to transitions in $\Delta^p \setminus \Delta^r$ as *maybe* transitions. All MTSs start in state 0, unless stated otherwise. Maybe transitions are denoted with a question mark following the label, and transitions on sets are short for an individual transition on every element of the set. We refer to the MTS (LTS) with a single state and an empty transition relation as the *empty* MTS (LTS). See Figure 1 for example MTSs.

Given an MTS $M = (S, L, \Delta^r, \Delta^p, s_0)$, M has a required transition on ℓ (denoted $M \xrightarrow{\ell}_r M'$) if $M' = (S, L, \Delta^r, \Delta^p, s_0')$ 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 $(s_0, \ell, s_0') \in \Delta^p$. We write $M \not\xrightarrow{\ell}$ when there is no M' such that $M \xrightarrow{\ell}_p M'$. For an MTS M, M_n denotes changing the initial state to n.

We capture the notion of elaboration of a partial description into a more comprehensive one using *refinement*.

DEFINITION 3. (Refinement) *Let* $\wp_M$ *be the universe of all MTSs. An MTS* N *is a* refinement *of an MTS* M*, written* $M \preceq N$*, if* $L_M = L_N$ *and* (M, N) *is contained in some refinement relation* $R \subseteq \wp_M \times \wp_M$ *for which the following holds for all* $\ell \in Act$*:*

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

Two MTSs are equivalent *(*$\equiv$*) if they refine each other.*

For a refinement relation R from M to N, we write $(s, t) \in R$ to denote $(M_s, N_t) \in R$. For example, $\mathcal{A}$ in Figure 1 is refined by $\mathcal{B}$ via the relation $\{(0, 0), (1, 1)\}$, since $\mathcal{B}$ can simulate the required behaviour in $\mathcal{A}$, and $\mathcal{A}$ can simulate the possible behaviour in $\mathcal{B}$.

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

DEFINITION 4. (Implementation) *An LTS* L *is an* implementation *of an MTS* M *if and only if* $M \preceq L$*.*

In this paper, we assume that all MTSs (and therefore LTSs) are *infinite-trace*:

DEFINITION 5. (Infinite-Trace) *An MTS* $M = (S_M, L_M, \Delta^r_M, \Delta^p_M, s_{0M})$ *is* infinite-trace *if for all* $s \in S_M$*, there exists* $a \in$ Act *and* $s' \in S_M$ *such that* $M_s \xrightarrow{a}_p M_{s'}$*.*

In other words, an MTS M is infinite-trace if every state has at least one outgoing transition. We call M *finite-trace* if it is not infinite-trace. The infinite-trace assumption is made because we intend to synthesize such models from linear temporal logic formulae, which are evaluated on infinite traces. From now on, we write "MTS" ("LTS") to mean an infinite-trace MTS (LTS), unless stated otherwise.

To put models of two different systems together, we use parallel composition.

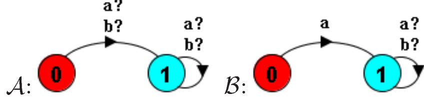

Figure 1: Example MTSs.

$$\text{TD}\ \frac{M \overset{\ell}{\dashrightarrow}_r M'}{M \| N \overset{\ell}{\dashrightarrow}_r M' \| N}\ \ell \notin \alpha N \qquad \text{MT}\ \frac{M \overset{\ell}{\dashrightarrow}_m M',\ N \overset{\ell}{\dashrightarrow}_r N'}{M \| N \overset{\ell}{\dashrightarrow}_m M' \| N'}$$

$$\text{MD}\ \frac{M \overset{\ell}{\dashrightarrow}_m M'}{M \| N \overset{\ell}{\dashrightarrow}_m M' \| N}\ \ell \notin \alpha N \qquad \text{TT}\ \frac{M \overset{\ell}{\dashrightarrow}_r M',\ N \overset{\ell}{\dashrightarrow}_r N'}{M \| N \overset{\ell}{\dashrightarrow}_r M' \| N'}$$

$$\text{MM}\ \frac{M \overset{\ell}{\dashrightarrow}_m M',\ N \overset{\ell}{\dashrightarrow}_m N'}{M \| N \overset{\ell}{\dashrightarrow}_m M' \| N'}$$

Figure 2: Rules for parallel composition.

Definition 6. (Parallel Composition) [19] *Let M and N be MTSs where $M = (S_M,\ L_M,\ \Delta^r_M,\ \Delta^p_M,\ s_{0M})$ and $N = (S_N,\ L_N,\ \Delta^r_N,\ \Delta^p_N,\ s_{0N})$. Then* parallel composition *($\|$) is a symmetric operator and $M \| P$ is the MTS $(S_M \times S_N,\ L_M \cup L_N,\ \Delta^r,\ \Delta^p,\ (s_{0M}, s_{0N}))$, where Δ^r and Δ^p are the smallest relations that satisfy the rules given in Figure 2.*

3. 3-VALUED FLTL

In this section, we first review the 3-valued Kleene logic [15] and use it to define a 3-valued variant of Fluent Linear Temporal Logic (FLTL) [6]. We then prove that FLTL properties are preserved in all implementations of a given MTS and define a model-checking procedure for this 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 $\bot$:

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

FLTL [6] 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 $a = \langle a, Act \setminus \{a\} \rangle$.

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), $\mathbf{F}$ (eventually), and $\mathbf{G}$ (always), as follows:

$$Fl \mid \neg\phi \mid \phi \vee \psi \mid \phi \wedge \psi \mid \mathbf{X}\phi \mid \phi\mathbf{U}\psi \mid \mathbf{F}\phi \mid \mathbf{G}\phi,$$

where $Fl \in \Phi$. For an infinite trace $\pi = a_0, a_1, a_2, \ldots$ over Act and some $i \in \mathbb{N}$, we write π^i for the suffix of π starting at a_i. Let Π be the set of infinite traces over Act.

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

Figure 3: Semantics for the satisfaction operator.

Definition 7. (Value of a Trace) *A trace $\pi = a_0, a_1, \ldots$ in Π is a* true trace *in M if there exists an infinite sequence $\{M_i\}$ 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 true 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 true trace in M. Finally, a trace π is a* false trace *in M if it is not a possible trace.*

For an MTS M, we denote by $\textsc{TrueTr}(M)$, $\textsc{MaybeTr}(M)$, $\textsc{PosTr}(M)$, and $\textsc{FalseTr}(M)$ the set of true, maybe, possible, and false traces over M, respectively. For example, in model $\mathcal{B}$ in Figure 1, the trace $a, b, b, \ldots$ is in $\textsc{MaybeTr}(\mathcal{B})$ (and thus in $\textsc{PosTr}(\mathcal{B})$) because $\mathcal{B}_1 \xrightarrow{b}_m \mathcal{B}_1$, and the trace $b, a, a, \ldots$ is in $\textsc{FalseTr}(\mathcal{B})$ because $\mathcal{B}_0 \overset{b}{\not\longrightarrow}$.

Given a trace $\pi \in \Pi$, a suffix 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 a trace π, the satisfaction operator $\models$ is defined as shown in Figure 3. The semantics for $\mathbf{F}$ and $\mathbf{G}$ is defined as $\pi \models \mathbf{F}\phi \triangleq \pi \models \mathbf{t}\ \mathbf{U}\ \phi$ and $\pi \models \mathbf{G}\phi \triangleq \pi \models \neg\mathbf{F}\neg\phi$. The definition in Figure 3 is standard [6] and yields a 2-valued operator.

The 3-valued semantics of FLTL over an MTS M is given by the function $\|\cdot\|^M$ that, for each formula $\phi \in$ FLTL, returns the value of ϕ in M.

Definition 8. (3-valued Semantics of FLTL) *The function $\|\cdot\|^M : \text{FLTL} \longrightarrow \mathbf{3}$ is defined as follows:*

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

As in [6], we consider only fair traces when evaluating an FLTL formula. When M is an LTS, our 3-valued semantics reduces to the standard 2-valued semantics of FLTL: ϕ is *true* in M (denoted $M \models \phi$) if every trace in $\textsc{TrueTr}(M)$ satisfies ϕ, and *false* in M (denoted $M \not\models \phi$) if there is a trace

in $\textsc{TrueTr}(M)$ that refutes ϕ. Otherwise, a formula ϕ evaluates to *maybe* in M if and only if no traces in $\textsc{TrueTr}(M)$ refute ϕ and there is at least one trace in $\textsc{PosTr}(M)$ that satisfies ϕ and one that refutes ϕ. For example, $\mathbf{F}a$ is *true* in $\mathcal{B}$ (see Figure 1) because every trace in $\textsc{PosTr}(\mathcal{B})$ satisfies $\mathbf{F}a$, whereas $\mathbf{F}c$ is *false* in $\mathcal{B}$ because every trace in $\textsc{PosTr}(\mathcal{B})$ refutes $\mathbf{F}c$. Finally, $\mathbf{F}b$ is *maybe* in $\mathcal{B}$ because $\textsc{TrueTr}(\mathcal{B})$ is empty and $a, b, b, \ldots$ is in $\textsc{MaybeTr}(\mathcal{B})$ (and therefore in $\textsc{PosTr}(\mathcal{B})$) and satisfies $\mathbf{F}b$, while $a, a, \ldots$ is in $\textsc{MaybeTr}(\mathcal{B})$ and refutes $\mathbf{F}b$.

The information ordering of this 3-valued variant of FLTL is preserved by refinement.

THEOREM 1. (Preservation of FLTL) *If M and N are MTSs, then* $M \preceq N \Rightarrow \forall \phi \in \text{FLTL} \cdot \|\phi\|^M \sqsubseteq_{inf} \|\phi\|^N$.

By Theorem 1 and Definition 4, if a property evaluates to *true* in M, it is *true* in all of its implementations, and if a property evaluates to *false* in M, it is *false* in all of its implementations. Furthermore, if a property evaluates to *maybe* in M, there is at least one implementation in which it is *true*, and one implementation in which it is *false*, which corresponds to the intuitive notion of a *maybe* property: it could be *true* of *false*. This semantics of a 3-valued logic is referred to as *thorough* [5].

Let M^+ be the LTS obtained from M by converting all maybe transitions to required, and M^- be the LTS obtained from converting all maybe transitions to false and removing all transitions that are not part of an infinite trace and all states that are not reachable from the initial state. In particular, M^+ represents $\textsc{PosTr}(M)$, while M^- represents $\textsc{TrueTr}(M)$, which is empty if and only if M^- is the empty LTS. The following result is adapted from [4].

THEOREM 2. (Model-checking) *For an MTS M:*

1. $\|\phi\|^M = \mathbf{t} \Leftrightarrow M^+ \models \phi$
2. $\|\phi\|^M = \mathbf{f} \Leftrightarrow \begin{cases} M^+ \models \neg\phi, \textit{ if } M^- \textit{ is the empty LTS} \\ M^- \not\models \phi \vee M^+ \models \neg\phi, \textit{ otherwise} \end{cases}$

and otherwise, $\|\phi\|^M = \bot$.

Condition 1 states that ϕ is *true* in M if and only if it is *true* in M^+, which follows from the fact that M^+ satisfies ϕ if and only if every trace in M^+ (and therefore every trace in $\textsc{PosTr}(M)$) satisfies ϕ. In Condition 2, $M^+ \models \neg\phi$ implies that every trace in $\textsc{PosTr}(M)$ satisfies $\neg\phi$, and therefore refutes ϕ. In addition, if M^- is not the empty LTS (i.e,. $\textsc{TrueTr}(M)$ is not empty), then $M^- \not\models \phi$ implies that there is a trace in $\textsc{TrueTr}(M)$ that refutes ϕ. Note that this is not the same as saying that $M^- \models \neg\phi$, which implies that every trace in $\textsc{TrueTr}(M)$ refutes ϕ. Putting these together coincides with the semantics in Definition 8 for $\|\phi\|^M = \mathbf{f}$. Hence, 3-valued FLTL model-checking reduces to classical FLTL model-checking, and can be supported by the LTSA tool [23].

4. SYNTHESIZING AN MTS

In this section, we describe the algorithm for synthesizing an LTS for a FLTL formula [21], and extend it to synthesize MTSs. We prove that the synthesized MTS captures all implementations that satisfy the formula, and show how to use merge to implement logical conjunction.

We restrict synthesis to safety properties [2], i.e., those properties that can be falsified by a finite sequence of events.

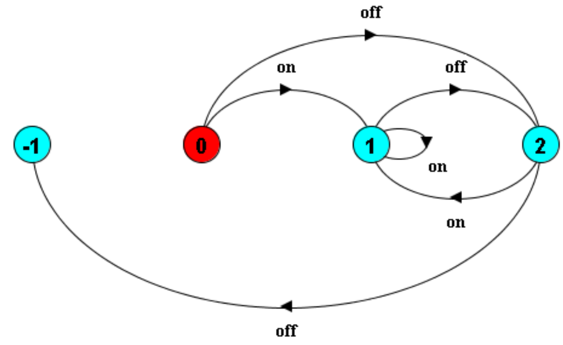

Figure 4: The property LTS for $\mathbf{G}(\mathbf{X}\ \textit{off} \Rightarrow \textit{LightOn})$.

Instead of handling liveness properties such as $\mathbf{F}a$ directly, 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. Thus, it suffices to provide bounded operators, such as $\mathbf{F}_{\leq q}$, which means eventually but in less than q time units [1], for capturing requirements using exclusively safety properties. This restriction is standard in requirements engineering approaches such as [31, 30]. We do not include new operators for handling bounded liveness in this presentation as they are simply syntactic sugar and can be defined in terms of standard linear time temporal operators. For instance, $\mathbf{F}_{\leq 2}\phi = \phi \vee \mathbf{X}\phi \vee \mathbf{XX}\phi$.

4.1 Synthesizing an LTS

The technique for model-checking an FLTL property ϕ over an LTS L involves constructing a Büchi automaton $B(\neg\phi)$ that recognizes all infinite traces over the alphabet that violate ϕ and checking that the synchronous product of $B(\neg\phi)$ with L is empty [6]. In this paper, we fix the alphabet to be the universe of actions, namely *Act*, so that synthesis always produces models with the same alphabet, regardless of whether the actions appear in ϕ (see Section 6 for a discussion regarding different alphabets). The Büchi automaton $B(\neg\phi)$ is *completed* by adding a sink state, and, for every state, adding a transition to the sink state on all actions that are not enabled in that state. Thus, $B(\neg\phi)$ has an execution for every infinite trace over the alphabet.

When ϕ is a safety property, there is only one accepting state in $B(\neg\phi)$, and this accepting state has only self-loop transitions, because safety properties are violated by a finite sequence of actions and a violation cannot be remedied. Thus, $B(\neg\phi)$ can be viewed as a *property* LTS for ϕ, i.e., an LTS with an error state, where the error state of the latter corresponds to the accepting state of the former with self-loops removed. All traces that reach the error state correspond to undesired behaviours, i.e., no infinite trace with a finite suffix that leads to the error state satisfies ϕ. For example, consider the property $\phi_{light} = \mathbf{G}(\mathbf{X}\textit{off} \Rightarrow \textit{LightOn})$, where $\textit{LightOn} = \langle \textit{on}, \textit{off} \rangle$, for a simple light switch. This property says that if the light can be turned off in the next state, then it is currently on, intended to express that the light can't be turned off if it is already off. The property

[1] In this paper, time is measured as the number of events that have occurred. This corresponds to the asynchronous interpretation of FLTL. In a synchronous semantics, time is given by the number of *tick* events that have elapsed. We have adopted the former for simplicity and because the latter can be encoded using asynchronous semantics [20].

LTS for ϕ_{light} is shown in Figure 4, where the error state is denoted by -1. The trace *off,off* leads to the error state and no infinite trace starting with *off,off* satisfies ϕ_{light}. For details on constructing a property LTS, see [6].

Once the property LTS has been constructed, all transitions not corresponding to an infinite trace are removed, followed by all states that are unreachable from the initial state (which always includes 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, and the finite traces that are removed correspond to all infinite traces in $B(\neg\phi)$ that refute ϕ. We denote by $L(\phi)$ the LTS generated by this procedure (e.g., $L(\phi_{light})$ is the LTS in Figure 4 with state -1 removed). By construction, $L(\phi)$ is *deterministic* (i.e., no state has more than one outgoing transition on the same action, called *non-determinism*) and infinite-trace.

Definition 9. (Satisfiability) *An* FLTL *formula* ϕ *is* satisfiable *if and only if there exists an LTS L such that* $L \models \phi$*; otherwise,* ϕ *is* unsatisfiable.

For example, no LTS satisfies $a \wedge \neg a$. For an unsatisfiable property ϕ, $L(\phi)$ is the empty LTS because all infinite traces refute ϕ, and thus all correspond to finite traces in the property LTS, which are removed.

For FLTL properties that are closed under stuttering [1], constructing an LTS for the conjunction of two properties can be done by first constructing the LTSs for the individual properties and composing them in parallel.

Theorem 3. (Composition, [21]) *If* ϕ_1 *and* ϕ_2 *are safety FLTL properties that are closed under stuttering:*

$$L(\phi_1)\|L(\phi_2) \equiv L(\phi_1 \wedge \phi_2).$$

The LTSA tool [23] has been extended to generate $L(\phi)$ when a desired FLTL property is specified using the keyword `constraint`. If this is not a safety property, an error message is generated. In addition, checking satisfiability of ϕ is done by checking if $L(\phi)$ is the empty LTS, since ϕ is satisfiable if and only if $L(\phi)$ is the empty LTS. If ϕ is unsatisfiable, a finite trace that leads to deadlock can be generated using the property LTS, helping to explain the cause.

4.2 Adding Partiality

The limitation of the algorithm in Section 4.1 is that the definite behaviour in $L(\phi)$ may not capture the behaviours which are truly necessary to satisfy ϕ. In other words, $L(\phi)$ rules out many other implementations that satisfy ϕ. For example, recall the light property ϕ_{light}. The system $L(\phi_{light})$ requires that in the initial state, the light can be switched off (i.e., $0 \xrightarrow{off}_r 2$). However, this behaviour is not required in order to satisfy this property: the LTS resulting from removing this transition also satisfies ϕ_{light}.

To address this limitation, we describe a simple procedure `generateMTS`(ϕ) for generating a partial model from a safety FLTL property ϕ. The procedure is as follows:

1. let $L = L(\phi)$ (constructed as in Section 4.1);
2. return $M(\phi)$, where $M(\phi)$ 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.

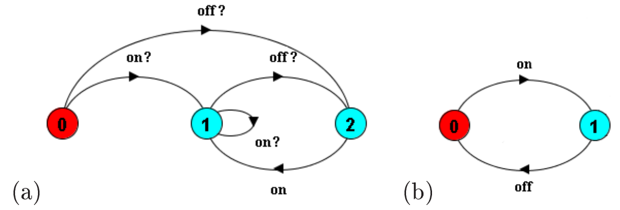

Figure 5: (a) The MTS $M(\phi_{light})$; (b) a possible implementation L_{light} of $M(\phi_{light})$.

Recall that for a satisfiable safety property ϕ, $L(\phi)$ contains all infinite traces that satisfy ϕ and no traces that refute ϕ. When a state in $L(\phi)$ has more than one outgoing transition, it indicates that there is more than one way to satisfy ϕ at that point in the trace. Thus, such transitions are not all necessary to satisfy ϕ, but any LTS that satisfies ϕ must contain at least one them. Such choices should be modelled with maybe behaviour instead of required behaviour, as in step 2. Also, if a state has only one outgoing transition, then because any implementation must be infinite-trace, this transition must be present in all implementations, and therefore should be modelled with required behaviour.

Theorem 4. (Correctness of `generateMTS`) *If* ϕ *is a satisfiable safety property, then* $\|\phi\|^{M(\phi)} = \mathbf{t}$.

Proof. $L(\phi)$ contains all traces that satisfy ϕ and no traces that refute ϕ. Therefore, every trace in $\text{PosTr}(M(\phi))$ satisfies ϕ and so $\|\phi\|^{M(\phi)} = \mathbf{t}$ by Definition 8. □

For example, $M(\phi_{light})$ is shown in Figure 5(a). The only required behaviour in this system is from state 2 to state 1 on action *on*, because state 1 is the only state from which the pump can be turned off after the first action in the trace, i.e., where $\mathbf{X}off$ is *true*. If $\mathbf{X}off$ is *false*, the property is satisfied trivially. This MTS allows for the possible implementation L_{light} (shown in Figure 5(b)), which prohibits the light from being turned on if it is already on, and turned off if it is already off. This implementation is constructed from $M(\phi_{light})$ via the refinement relation {(0,0),(1,1),(2,0)}.

The MTS $M(\phi)$ constructed for ϕ is *minimal*, i.e., every other MTS that satisfies ϕ refines it.

Theorem 5. (Minimality of $M(\phi)$) *If* ϕ *is a satisfiable safety property, then:*

$$\forall M \in \wp_M \cdot L_M = L_{M(\phi)} \wedge \|\phi\|^M = \mathbf{t} \Rightarrow M(\phi) \preceq M.$$

Proof. We show how to build a refinement relation R between $M(\phi)$ and M. Let $(M(\phi), M) \in R$, i.e., assume that initial states are related. Suppose $M(\phi) \xrightarrow{a}_r M(\phi)'$ for some $a \in Act$. Then $M(\phi) \not\xrightarrow{b}$ for any $b \neq a$ by step 2 of `generateMTS`. Because $\|\phi\|^M = \mathbf{t}$, every trace in $\text{PosTr}(M)$ satisfies ϕ by Definition 8, and thus every trace must start with a; otherwise, $L(\phi)$ would have more than one outgoing transition, and hence so would $M(\phi)$. Thus, there exists M' such that $M \xrightarrow{a}_r M'$, so $(M(\phi)', M') \in R$. It follows that Condition (1) in Definition 3 holds.

$M(\phi)$ contains all traces that satisfy ϕ, and therefore $\text{PosTr}(M) \subseteq \text{PosTr}(M(\phi))$. In addition, for an action a_i in π, there exists $M(\phi)_i$ and $M(\phi)_{i+1}$ such that $M(\phi)_i \xrightarrow{a_i}_p M(\phi)_{i+1}$, and there is no $b \neq a_i$ such that $M(\phi)_i \xrightarrow{b}_r$, because step 2 of `generateMTS` would have converted it to a

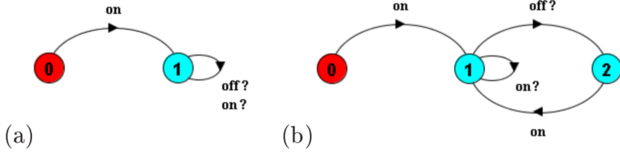

Figure 6: (a) $M(\textbf{\textit{LightOn}})$ **; (b)** $M(\phi_{\textbf{\textit{light}}}) +_\infty M(\textbf{\textit{LightOn}})$.

maybe transition. Thus, the sequence of states in M corresponding to π can be identified in R with the sequence of states in $M(\phi)$ corresponding to π, and therefore Condition (2) in Definition 3 holds. ☐

Hence, by Theorems 1 and 5, given a safety property ϕ and an LTS L (with the same alphabet as $M(\phi)$), $L \models \phi$ if and only if L is an implementation of $M(\phi)$, i.e., $L \in \mathcal{I}(M(\phi))$. Note that although $M(\phi)$ is deterministic, it characterizes the non-deterministic implementations.

4.3 Logical Conjunction via Merge

Consider the MTS for ϕ_{light} in Figure 5(a). This system allows event *off* to occur in the initial state. In other words, ϕ_{light} does not imply that the light can only be switched off if it is already on, as intended. Instead of strengthening ϕ_{light} and generating a new MTS, it can be more practical to compose a different model with $M(\phi_{light})$ such that the composition satisfies the desired requirements, similar to Theorem 3.

Parallel composition does not suffice because it does not preserve temporal properties, and in particular FLTL properties, since the composition may not be a refinement of either operand. For example, rule MT in Figure 2 states that if a required transition in M is synchronized with a maybe transition in N, then $M \| N$ has a maybe transition, and hence does not refine M.

Instead, for MTSs, composition that preserves temporal properties takes the form of *merge* [27]. Merging consistent (finite- or infinite-trace) MTSs M and N (denoted $M + N$) is about finding a *common refinement*, i.e., a model that refines both systems, where consistency is defined as the existence of a common refinement. In [27], it is argued that the merged model should not introduce unnecessary behaviours, and therefore should be the *least common refinement*. Formally, P is the least common refinement (LCR) of M and N if P is a common refinement of M and N and for all common refinements Q of M and N, $P \preceq Q$. Although the LCR does not always exist, it does exist for consistent deterministic MTSs with equal alphabets [3].

In this subsection, we aim to use merge as logical conjunction for FLTL formulae, i.e., to prove that $M(\phi_1 \wedge \phi_2) \equiv M(\phi_1) + M(\phi_2)$. Just as we remove finite traces from $L(\phi)$ in Section 4.1, we use an augmented merge operator $+_\infty$ such that $M(\phi_1) +_\infty M(\phi_2)$ is obtained from $M(\phi_1) + M(\phi_2)$ by removing transitions not corresponding to an infinite trace and then all unreachable states. From now on, when we say *merge*, we mean the operator $+_\infty$. The main difficultly is that the merge of two consistent infinite-trace MTSs constructed from properties may be the empty LTS, for which FLTL formulae are undefined. For example, consider the MTSs for $\phi_1 = \mathbf{G}(a \vee b)$ and $\phi_2 = \mathbf{G}(c \vee d)$ shown in Figure 7(a) and (b), respectively. The empty LTS is a common refinement of $M(\phi_1)$ and $M(\phi_2)$ and is not infinite-trace. Fortunately, the cases when $+_\infty$ produces the empty LTS correspond to an unsatisfiable property:

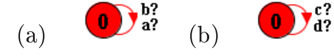

Figure 7: (a) $M(\mathbf{G}(a \vee b))$ **; (b)** $M(\mathbf{G}(c \vee d))$.

Theorem 6. (Finite-Trace Merge) *If $M(\phi_1)$ and $M(\phi_2)$ are consistent and $M(\phi_1) +_\infty M(\phi_2)$ is the empty LTS, then $\phi_1 \wedge \phi_2$ is unsatisfiable.*

In particular, property ϕ_1 above requires that either a or b always occur, and ϕ_2 requires that either c or d always occur, and therefore no LTS satisfies $\phi_1 \wedge \phi_2$, i.e., $\phi_1 \wedge \phi_2$ is unsatisfiable. In particular, Theorem 6 provides an effective procedure for determining the satisfiability of $\phi_1 \wedge \phi_2$, and is an alternative to constructing $M(\phi_1 \wedge \phi_2)$ and then checking if it is the empty LTS.

Theorem 6 also provides the basis for logical conjunction, because if $\phi_1 \wedge \phi_2$ is satisfiable and $M(\phi_1)$ and $M(\phi_2)$ are consistent, then $M(\phi_1) +_\infty M(\phi_2)$ is infinite-trace. In addition, recall that $M(\phi_1)$ and $M(\phi_2)$ have the same alphabet, and therefore if $M(\phi_1)$ and $M(\phi_2)$ are consistent, their LCR exists because they are deterministic. Therefore, for a satisfiable safety property $\phi_1 \wedge \phi_2$, building the MTS for $\phi_1 \wedge \phi_2$ using `generateMTS` is equivalent to building the MTSs for ϕ_1 and ϕ_2 individually and then merging them.

Theorem 7. (Conjunction) *If $\phi_1 \wedge \phi_2$ is a satisfiable FLTL safety property, then $M(\phi_1 \wedge \phi_2) \equiv M(\phi_1) +_\infty M(\phi_2)$.*

Proof. $\|\phi_1\|^{M(\phi_1 \wedge \phi_2)} = \mathbf{t}$ and $\|\phi_2\|^{M(\phi_1 \wedge \phi_2)} = \mathbf{t}$ implies that $M(\phi_1) \preceq M(\phi_1 \wedge \phi_2)$ and $M(\phi_2) \preceq M(\phi_1 \wedge \phi_2)$ by Theorem 5. Therefore $M(\phi_1 \wedge \phi_2)$ is a common refinement of $M(\phi_1)$ and $M(\phi_2)$, and thus $M(\phi_1) +_\infty M(\phi_2) \preceq M(\phi_1 \wedge \phi_2)$ by definition of $+_\infty$.

By the previous argument, $M(\phi_1)$ and $M(\phi_2)$ are consistent and since $\phi_1 \wedge \phi_2$ is satisfiable, $M(\phi_1) +_\infty M(\phi_2)$ is infinite-trace by Theorem 6. Thus $\|\phi_1 \wedge \phi_2\|^{M(\phi_1)+_\infty M(\phi_2)} = \mathbf{t}$ by Theorem 1 and the definition of $+_\infty$. By Theorem 5, $M(\phi_1 \wedge \phi_2) \preceq M(\phi_1) +_\infty M(\phi_2)$. ☐

Theorem 7 is the 3-valued counterpart to Theorem 3, and does not require ϕ_1 and ϕ_2 to be closed under stuttering.

For an example of Theorem 7, recall the light property ϕ_{light}. The problem with $M(\phi_{light})$ in Figure 5(a) is that not every trace begins with *on*, i.e., not every trace satisfies the fluent *LightOn*. The MTS $M(\mathit{LightOn})$ is shown in Figure 6(a), and is consistent with $M(\phi_{light})$. Therefore, their merge exists (see Figure 6(b)), and the merged model satisfies both properties and has the desired behaviours with respect to turning the light off: it cannot be turned off if it is already off.

The other direction of Theorem 7 does not hold, i.e., $M(\phi_1) +_\infty M(\phi_2) \equiv M(\phi_1 \wedge \phi_2)$ does not imply that $\phi_1 \wedge \phi_2$ is satisfiable, since by Theorem 6, if $M(\phi_1)$ and $M(\phi_2)$ are consistent but their merge is the empty LTS, then $\phi_1 \wedge \phi_2$ is unsatisfiable. However, we have the following desirable result with respect to infinite-trace merges:

Theorem 8. (Satisfiability for Infinite-Trace Merges) *If MTSs $M(\phi_1)$ and $M(\phi_2)$ are consistent and $M(\phi_1) +_\infty M(\phi_2)$ is infinite-trace, then $\phi_1 \wedge \phi_2$ is satisfiable.*

Proof. $L \models \phi$ for any L in $\mathcal{I}(M(\phi_1) +_\infty M(\phi_2))$, and $\mathcal{I}(M(\phi_1) +_\infty M(\phi_2))$ is clearly non-empty. ☐

Figure 8: $M(\phi_1)$ for the mine pump system.

Putting Theorems 6, 7, and 8 together, $M(\phi_1)$ and $M(\phi_2)$ are consistent and $M(\phi_1) +_\infty M(\phi_2)$ is infinite-trace if and only if $\phi_1 \wedge \phi_2$ is satisfiable. In other words, merging in the realm of infinite-trace MTSs is precisely logical conjunction.

5. CASE STUDY: THE MINE PUMP

In this section, we illustrate our techniques on a pump controller system in a mine sump [16]. 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 active when there is no methane gas present in the mine. The pump controller monitors the water and methane levels by communicating with two sensors, and controls the pump in order to guarantee the safety properties of the pump system.

Properties. To ease presentation, we give a simplified version of the mine pump system in which there are three water levels (low, medium, and high), and consider only a subset of the desired properties of the pump controller.

The properties of the mine pump controller rely on the following fluents:

$$\begin{array}{lcl} HighWater & = & \langle highWater, \{lowWater, medWater\}\rangle \\ LowWater & = & \langle lowWater, \{medWater, highWater\}\rangle \\ PumpOn & = & \langle switchOn, switchOff\rangle \\ Methane & = & \langle methAppears, methLeaves\rangle \end{array}$$

where $LowWater$ is initially *true* and the rest are initially *false*. The following are several well known safety properties of the mine pump controller:

$$\begin{array}{lcl} \phi_1 & = & \mathbf{G}(tick \Rightarrow (HighWater \wedge \neg Methane \Rightarrow PumpOn)) \\ \phi_2 & = & \mathbf{G}(tick \Rightarrow (LowWater \Rightarrow \neg PumpOn)) \\ \phi_3 & = & \mathbf{G}(tick \Rightarrow (Methane \Rightarrow \neg PumpOn)) \end{array}$$

These properties make use of an explicit *tick* event, signalling the successive ticks of a global clock to which each timed component synchronizes. This corresponds to a standard approach to modelling discrete-time in event-based formalisms [23]. Modelling time is required for systems like the mine pump, where urgency of certain events, such as switching the pump off to avoid an explosion, must be captured. Specifically, property ϕ_1 expresses that if there is high water and no methane when a *tick* occurs, the pump should be on. Property ϕ_2 expresses that if there is low water when a *tick* occurs, the pump must be off, and similarly, property ϕ_3 requires the pump to be off if there is methane when a *tick* occurs. Note that properties ϕ_1 to ϕ_3 are simplifications of more realistic requirements that would make the presentation of this case study too complex. For instance, a more realistic version of ϕ_1 is one which states that if the water level is high and there is no methane until the next time unit, then the pump is on before the next time unit. The formalization of this property in FLTL is $\mathbf{G}((HighWater \wedge \neg Methane)\mathbf{U}tick \Rightarrow \neg tick\ \mathbf{U}PumpOn)$.

Model Generation. To generate an MTS that characterizes the implementations that satisfy ϕ_1 – ϕ_3, we let $\phi = \phi_1 \wedge \phi_2 \wedge \phi_3$ and call `generateMTS`(ϕ) to generate $M(\phi)$. This model satisfies ϕ, and therefore ϕ is consistent. Due to the size of $M(\phi)$ (12 states, 92 transitions), we limit our discussion to the model $M(\phi_1)$ (8 states, 38 transitions) shown in Figure 8, while noting that the same reasoning applies to $M(\phi)$. In Figure 8, a transition on *notHigh* is an abbreviation for a transition on either *lowWater* and *medWater*.

$M(\phi_1)$ predominantly contains maybe behaviours due to the nature of ϕ_1. In particular, ϕ_1 states that if there is high water and no methane when a *tick* event occurs, the pump must be on. This does not lead to a required transition on *switchOn* when there is high water and no methane, because other events may happen before the occurrence of *tick*. For

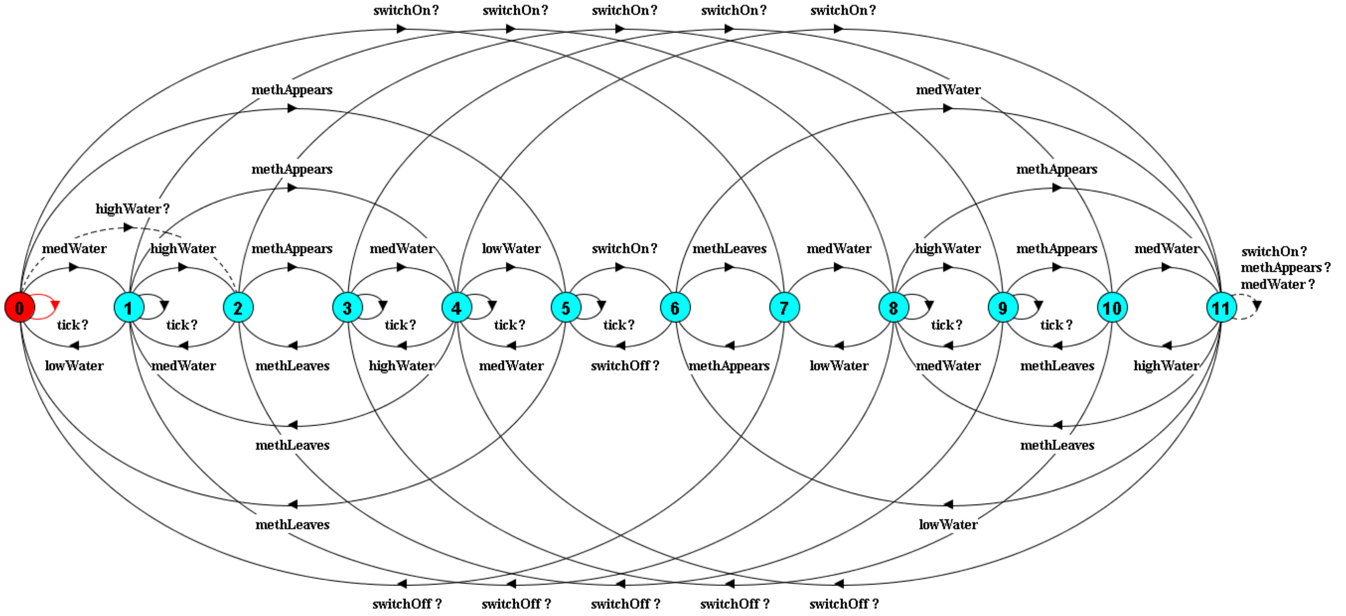

Figure 9: A refinement of $M(\phi)$ (dotted transitions are example maybe transitions refined to false).

example, after following the transition from state 0 to state 7 on *highWater*, the water is high and there is no methane, but the *tick* event cannot occur until the pump is on (state 7 to state 5 on *switchOn*) or there is no longer high water or methane appears (state 7 to state 0 on *notHigh* and state 7 to state 4 on *methAppears*), in which case the pump isn't required to be on. In other words, ϕ_1 does not require immediacy in the sense that *switchOn* must be the next action to occur. In addition, ϕ_1 specifies sufficient conditions for the pump to be on, but leaves open whether the pump should also be on at low or medium water, which leads to maybe transitions on *switchOn* (e.g., from state 0 to state 1). Finally, properties of the environment (e.g., "methane cannot appear if it is already present") are not in the scope of ϕ_1, which leads to maybe self-loops (e.g., on *methAppears*).

Analysis. Having synthesized an unbiased operational model $M(\phi)$ for the pump controller from a set of declarative system properties, it is possible to use this model to validate the intended system behaviour using techniques that require executable models, such as animation and simulation. For instance, the SceneBeans animation toolkit [24] could be used for enabling graphical animations of MTSs, possibly using a visual convention to distinguish between maybe and required behaviours. In addition, visualization of the resulting MTS may provide feedback, particularly in the case of small models, such as assurance that there has not been an error while formulating the requirements in temporal logic. For larger models, visualization of an abstraction of the MTS (using action hiding and minimization [23]), or some conversion into a hierarchical representation, such as Statecharts [8], may be adequate.

In addition, model-checking the synthesized model can also provide useful feedback concerning the intended system behaviour. For example, consider the property

$$\phi_4 \quad = \quad \mathbf{G}(switchOn \Rightarrow HighWater),$$

which expresses that there should be high water whenever the pump is switched on. This property evaluates to *maybe* in $M(\phi_1)$ due to the transition from 0 to 1 on *switchOn*, and similarly evaluates to *maybe* in the full model $M(\phi)$. This indicates that the properties leave open whether the pump can be turned on at a low or medium water level. A model-checker can return an explanation why the property is not *true*, i.e. a possible trace that refutes ϕ_4. Understanding such traces can then be aided by the animation and simulation techniques discussed above.

Formula ϕ_4 is not the only property that is left open by $M(\phi)$. By Theorem 5, we know that $M(\phi)$ is the least refined MTS that satisfies ϕ, which means that it does not bias the possible implementations for the pump controller by, for instance, making arbitrary decisions on whether the pump should or should not be turned off at medium water.

$M(\phi)$ is less refined than models that have been previously constructed from analogous properties by hand (e.g., [3]) and automatically (e.g., [20]), which indicates that hidden assumptions were made at modelling or synthesis time in both approaches. In [20], LTSs are used to model the mine pump, and thus must pick one of the possible implementations for the initial set of requirements. The LTS synthesized by that approach is a "greedy" implementation that attempts to perform as much as possible without violating any properties. In [3], MTS models are provided for the mine pump that are intended to satisfy properties analogous to ϕ_1 – ϕ_3. However, the resulting model has a number of required transitions on water level and methane actions. They are due to the implicit assumption that the pump controller is required to never constrain the occurrence of actions controlled by the environment, hence providing a required transition on every state for every water level and methane action. While this is a common assumption, we believe that the ability to introduce it explicitly (or not introduce it at all) aids the synthesis process.

Below, we discuss how an MTS may be refined iteratively to narrow down the number of acceptable implementations, and eventually reaching one LTS. We exemplify this elabora-

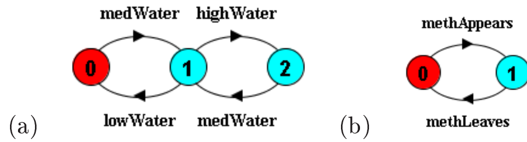

Figure 10: (a) An MTS describing the water level environment; (b) An MTS describing the methane environment.

tion by iteratively refining $M(\phi)$. To illustrate the fact that $M(\phi)$ is unbiased, as opposed those in [3] and [20], we show how to refine $M(\phi)$ into implementation that makes different choices than those made in [20], yet satisfies property ϕ.

Elaboration. As can be seen for $M(\phi_1)$ in Figure 8, the initial synthesized model can have many maybe transitions due to the declarative nature of properties. Therefore, in addition to validating the synthesized MTS, support is needed for iterative refinement, which removes maybe transitions, narrowing down the acceptable implementations and leading eventually to one LTS implementation.

MTS merging provides an elegant and automated way of refining an existing partial model. Merging guarantees (by Theorem 1 and the fact that it builds common refinement [27]) that properties of both models being merged are preserved in all implementations of the resulting MTS. Consequently, if, for instance, by replaying the behaviour of an MTS to users, a new property becomes apparent, that property can be integrated with the existing MTS description of the system by first synthesizing an MTS for the property and then merging it with the rest of the system. Additional information can also come in the form of a hand-crafted MTS (or LTS) that can then be merged in.

For example, consider the model shown in Figure 9. To obtain this model through successive refinements of $M(\phi)$, first we synthesize an MTS for the conjunction of the following two properties, merging it with $M(\phi)$:

$$\begin{aligned} \phi_5 &= \mathbf{G}(\mathbf{X}\ \mathit{switchOn} \Rightarrow \neg \mathit{PumpOn}) \\ \phi_6 &= \mathbf{G}(\mathbf{X}\ \mathit{switchOff} \Rightarrow \mathit{PumpOn}) \wedge \neg \mathit{switchOn}, \end{aligned}$$

which state that the pump can only be turned on if it is already off (ϕ_5), and the pump can only be turned off if it is already on (ϕ_6). These properties exploit FLTL to express the relation between the occurrence of an event (recall that the fluent *switchOn*, which should be interpreted as "the event *switchOn* has just occurred", is defined implicitly as $\langle \mathit{switchOn}, \mathit{Act} \setminus \{\mathit{switchOn}\}\rangle$). This step does not refine all maybe transitions on *switchOn* and *switchOff* to required transitions because $M(\phi_5 \wedge \phi_6)$ contains self-loops on other actions in the alphabet, but rather restricts the occurrence of consecutive *switchOn* and *switchOff* events.

Second, we construct two models of the environment by hand: one that contains assumptions on the water levels (Figure 10(a)) and one that contains assumptions on the methane (Figure 10(b)), with the goal to restrict implementations to those that make the above assumptions on their environment. We then merge the models in Figure 10 with $M(\phi_5 \wedge \phi_6) + M(\phi)$ (the result of our previous step). Although these models do not have the same alphabet, by results in [3], a unique merge exists and the properties are preserved by Theorem 1.

The resulting model in Figure 9 is a refinement of $M(\phi)$. Note that in Figure 9, for illustration purposes, we depict examples of transitions from $M(\phi)$ that were refined to false using dotted lines. In particular, this refinement requires that the pump is only switched on if it is already off and only switched off if it is already on, removing unnecessary maybe self-loops on *switchOn* and *switchOff* (e.g., in state 11 on *switchOn*). In addition, methane cannot appear if it has already appeared and cannot leave if it has already left, removing unnecessary maybe self-loops on *methAppears* and *methLeaves* (e.g., in state 11 on *methAppears*). Finally, the water level cannot move from low to high or from high to low without going through medium, and cannot move from the current level to the current level (e.g., low to low). This removes transitions that skip over medium water (e.g., the maybe transition from state 0 to state 2 on *highWater*), as well as maybe self-loops on the water levels (e.g., in state 11 on *medWater*).

Additionally, scenario-driven approaches can be used as an alternative way of pruning the possible implementations described by an MTS. That is, positive and negative examples of system behaviour provided by a user could be used to refine maybe transitions to true or false ones. For example, refer to the model in Figure 9. Property ϕ_4, which expresses that the water is only switched on when there is high water, evaluates to *maybe* because of maybe transitions from 0 to 7 and from 1 to 8 on *switchOn*. If the behaviours expressed by this property are provided by scenarios, then by using the animation and walk-through techniques discussed above, this transition can be refined to *false*. Such refinements make the resulting model inconsistent with that in [20], since the model in [20] requires that the pump can be turned on at low and medium water.

An example of how such scenario-driven elaboration could be done in an interactive and iterative fashion is as follows: The MTS can be animated by a user starting at the initial state and choosing which enabled transition to take. Taking a transition leads to a new current state which puts forward a new set of enabled transitions for the user to chose from. Maybe transitions that are fired by the user can be converted into required transitions, while a simple interface which distinguishes maybe from required transitions could allow a user to indicate that certain maybe transitions should not be available in the current state. Such an indication would lead to refining the selected maybe transition to false. To aid the user, a graphical animation of the MTS, depicting the state of the system, could be displayed while the user drives the understanding and refinement process.

Summary. In this section, we have shown how our synthesis approach can be used to build an operational model of the mine pump controller that satisfies all given properties without biasing the possible implementations of the controller. In particular, the synthesized model does *not* constrain implementations that satisfy the given properties. We compared the synthesized model $M(\phi)$ to existing models for the mine pump controller [3, 20] and exemplified how our model does not introduce implementation bias and can be refined into one of the implementations that satisfies ϕ.

6. DISCUSSION AND RELATED WORK

In this section, we discuss our results and decisions we have made, comparing our approach with related work.

MTSs over Infinite Traces. In Linear Temporal Logic

(LTL) [25] and thus 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 [11], or LTL semantics is extended to enable explicit reasoning about such traces [9]. 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 1 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 that a transition on either label can be refined to false in the implementation, as long as at least one transition becomes 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 1).

3-valued FLTL and Model Synthesis. In Section 3, 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 [7], enabling Theorem 2.

The relation between MTSs and 3-valued modal μ-calculus logics [3, 12] 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 [6, 31, 13]. 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 [6]. 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 4 restricts the synthesis of MTSs to safety properties, implementations obtained via refinement will preserve all properties (including liveness) that the synthesized MTS may have.

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 possible, 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 observational refinement [27, 3], as we explain below.

The synthesized MTS $M(\phi)$ is refined by any LTS that satisfies ϕ and has the same alphabet as $M(\phi)$, by Theorem 5. If $L \models \phi$ but L has a different alphabet than $M(\phi)$, it is possible to restrict the alphabets of L and $M(\phi)$ by replacing actions that are not shared by both with *silent* actions (i.e., actions that are unobservable by the environment of the system). In this case, L is an *observational* refinement of $M(\phi)$, with the appropriate alphabet restrictions, where observational refinement is effectively refinement that ignores differences in silent transitions.

Merge can also serve as logical conjunction when considering models with different alphabets: If $M(\phi_1)$ and $M(\phi_2)$ are constructed using the alphabets defined by the fluents contained in ϕ_1 and ϕ_2, then, if their LCR exists, merging becomes the logical conjunction given that the alphabet of the merged model is appropriately restricted. Thus, our approach naturally generalizes to implementations with augmented alphabets.

Related Work. A number of approaches to building event-based models from properties exist [18, 29, 26, 13, 21, 22]. For instance, [18] have developed a technique for automatically translating a goal-oriented requirements model into a tabular event-based specification in the form of SCR [10]. [29, 26] have developed Statechart [8] synthesis techniques to support animation and validation of property-based specifications. In [13], Formal Tropos goal models are translated into the event-based specification language Promela for verification using the SPIN tool [11]. All of these approaches, as well as [21], build *one* of the many possible event-based models that satisfy the given properties. An alternative approach, presented in [22], requires a set of properties to be so strong as to allow a unique implementation.

Operational models have also been built from scenario descriptions [28, 17]. 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.

7. SUMMARY AND FUTURE WORK

In this paper, we have presented an automated technique for constructing partial behavioural models in the form of MTSs from safety properties expressed in FLTL. The resulting models do not introduce implementation bias, leaving any behaviour neither required nor prohibited by requirements as behaviour that may or may not be part of a final

LTS implementation. These MTS models can be refined into a desired implementation using a number of elaboration techniques.

In addition to the synthesis, the paper has presented a 3-valued version of FLTL which can be evaluated over partial behaviour models such as MTSs. A number of results relating the logic, MTS synthesis, and MTS merge support compositional model elaboration and consistency checking. We have illustrated the use of our synthesis approach and the utility of the formal results using a mine pump example.

Our long-term goal is to provide a sound engineering approach to the development of software systems via automated support for constructing and elaborating behavioural models incrementally from scenario-based and declarative behaviour specifications, starting with partial behaviour models and leading to behaviour models that fully describe the required system behaviour. We therefore plan to extend our approach to accomodate scenario descriptions, both to synthesize partial behaviour models and to support their refinement.

Key to success of the approach presented in this paper is to provide adequate support for model elaboration, from the constructed highly partial model, into a desired implementation. To this end, we plan to further develop and implement the methodology for model elaboration which combines visualization strategies, support for elicitation and application of domain assumptions and requirements. We also intend to conduct larger case studies that illustrate our model synthesis and elaboration approach.

8. REFERENCES

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

[2] B. Alpern and F. Schneider. "Defining Liveness". *Information Processing Letters*, 21:181–185, 1985.

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

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

[5] G. Bruns and P. Godefroid. "Generalized Model Checking: Reasoning about Partial State Spaces". In *CONCUR'00*, volume 1877 of *LNCS*, pages 168–182, 2000.

[6] D. Giannakopoulou and J. Magee. "Fluent Model Checking for Event-Based Systems". In *ESEC/FSE'03*, pages 257–266, 2003.

[7] A. Gurfinkel and M. Chechik. "How Thorough is Thorough Enough". In *CHARME'05*, volume 3725 of *LNCS*, pages 65–80, 2005.

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

[9] K. Havelund and G. Rosu. "Monitoring Programs Using Rewriting". In *ASE'01*, pages 135–143, 2001.

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

[11] G. Holzmann. "The Model Checker SPIN". *IEEE TSE*, 23(5):279–295, May 1997.

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

[13] R. Kazhamiakin, M. Pistore, and M. Roveri. "Formal Verification of Requirements using SPIN: A Case Study on Web Services". In *SEFM'04*, pages 406–415, 2004.

[14] R. Keller. "Formal Verification of Parallel Programs". *CACM*, 19(7):371–384, 1976.

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

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

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

[18] R. D. Landtsheer, E. Letier, and A. van Lamsweerde. "Deriving Tabular Event-Based Specifications from Goal-Oriented Requirements Models". In *RE'03*, pages 200–210, 2003.

[19] K. Larsen and B. Thomsen. "A Modal Process Logic". In *LICS'88*, pages 203–210, 1988.

[20] E. Letier, J. Kramer, J. Magee, and S. Uchitel. "Fluent Temporal Logic for Discrete-time Event-Based Models". In *FSE'05*, pages 70–79, 2005.

[21] E. Letier, J. Kramer, J. Magee, and S. Uchitel. "Deriving Event-Based Transition Systems from Goal-Oriented Requirements Models". Technical report, Imperial College London, April 2006. http://www.doc.ic.ac.uk/ su2/.

[22] E. Letier and A. van Lamsweerde. "Deriving Operational Software Specifications from System Goals". In *FSE'02*, pages 119–128, 2002.

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

[24] J. Magee, N. Pryce, D. Giannakopoulou, and J. Kramer. "Graphical Animation of Behavior Models". In *ICSE'00*, pages 499–508, 2000.

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

[26] C. Ponsard, P. Massonet, A. Rifaut, J. Molderez, A. van Lamsweerde, and H. T. Van. "Early Verification and Validation of Mission-Critical Systems". In *FMICS'04*, 2004.

[27] S. Uchitel and M. Chechik. "Merging Partial Behavioural Models". In *FSE'04*, pages 43–52, 2004.

[28] S. Uchitel, J. Kramer, and J. Magee. "Synthesis of Behavioural Models from Scenarios". *IEEE TSE*, 29(2):99–115, 2003.

[29] H. T. Van, A. van Lamsweerde, P. Massonet, and C. Ponsard. "Goal-Oriented Requirements Animation". In *RE'04*, pages 218–228, 2004.

[30] A. van Lamsweerde, R. Darimont, and E. Letier. "Managing Conflicts in Goal-Driven Requirements Engineering". *IEEE TSE*, 24(11):908–926, 1998.

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