Behaviour Model Synthesis From Properties and Scenarios

Sebastian Uchitel[†] **Greg Brunet**[†‡] **Marsha Chechik**[‡]

[†]Department of Computing
Imperial College, London, SW7 2RH, UK
{s.uchitel,gwb}@doc.ic.ac.uk

[‡]Department of Computer Science
University of Toronto, Toronto, ON, Canada
{gbrunet,chechik}@cs.toronto.edu

Abstract

Synthesis of behaviour models from software development artifacts such as scenario-based descriptions or requirements specifications not only helps significantly reduce the effort of model construction, but also provides a bridge between approaches geared toward requirements analysis and those geared towards reasoning about system design at the architectural level. 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.

1 Introduction

Event-based behavioural models such as Labelled Transition Systems (LTSs) 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. In particular, synthesis from declarative requirements specifications (e.g., [11, 13, 21, 16, 8]) or from scenarios and use cases (e.g., [10, 20, 9, 3]), has been studied extensively.

Synthesis from declarative specifications such as goal models describing the requirements of a system 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, the another that is well suited for architectural and high-level design analysis. Properties can be thought of as statements that prune the space of acceptable behaviours of the system to be. 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.

Synthesis from scenario-based specifications such as Message Sequence Charts (MSCs) [7] has a number of advantages that complement those of property synthesis. 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. By synthesizing behaviour models from scenarios, it is possible to support early analysis, validation, and incremental elaboration of behaviour models. 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 this paper, we argue that classical state machine models such as LTSs are insufficiently expressive to support synthesis from *both* properties and scenarios. We extend existing LTS synthesis algorithms to produce Modal Transition Systems (MTSs) [12] 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: (*i*) a technique for automatically generating MTSs from safety properties expressed in

Fluent Linear Temporal Logic (FLTL); (*ii*) a technique for extending LTS synthesis from scenario approaches to support construction of MTS models; (*iii*) a demonstration that composition of MTSs is a *merge* [19], and therefore it preserves the original properties and scenarios and completes the approach for combined synthesis of behaviour models from all these artifacts. It also characterizes all MTSs (and LTSs) that preserve these artifacts and consequently represents the starting point for further elaboration of system behaviour. (*iv*) 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 requirements elaboration process.

The rest of the paper is organized as follows. We begin with a motivating example in Section 2, followed by the necessary background in Section 3 and discussion of FLTL in Section 4. Sections 5 and 6 present algorithms for synthesizing MTSs from safety properties and scenarios, respectively. In Section 7, we use the merge operator to put such partial behavioural descriptions together. In Section 8, 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 9, and conclude in Section 10.

2 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. Fig. 1 provides some examples of the intended system behaviour using a standard message sequence chart notation [7]. The scenario sc describes a repetition (the outer *rep* box) of a choice (the inner *alt* box) between two sequences of actions: (1) a User requests authentication from 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* . . . .

The Webmail system is required to enforce legal and private access to the emails it stores. These requirements are formalized in FLTL [4] in Fig. 2 as properties p_1 and p_2. Legal access requires the User be *Registered* if it is to be *LoggedIn*. Private access requires that the User be *LoggedIn* if it is to receive e-mail from the Server (*sendMsg*). *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 been *disabled*. An additional requirement, p_3, specifies that users should be sent an acknowledgment on logout.

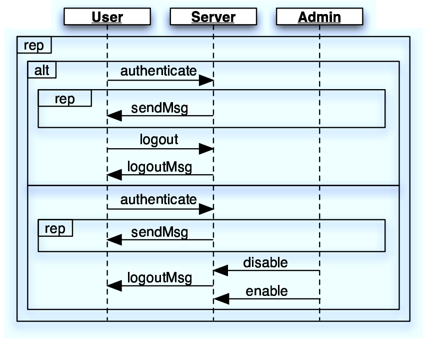

Figure 1. Webmail scenario specification sc.

$$\begin{array}{l} \mathit{Registered} = \langle \mathit{enable}, \mathit{disable} \rangle \text{ initially } \mathit{TRUE} \\ \mathit{LoggedIn} = \langle \mathit{authenticate}, \{\mathit{logout}, \mathit{disable}\} \rangle \text{ initially } \mathit{FALSE} \end{array}$$

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

Figure 2. Webmail system properties.

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 [4], an LTS model $L(P)$ shown in Fig. 3. This model describes *all* possible behaviours over the events Act_{web} = $\{\mathit{enable}, \mathit{disable}, \mathit{authenticate}, \mathit{logout}, \mathit{sendMsg}, \mathit{logoutMsg}\}$ 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 $L(P)$.

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 Fig. 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 extended 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

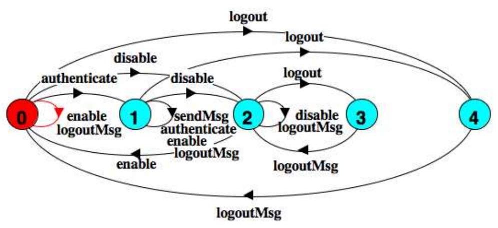

Figure 3. LTS synthesized from property P.

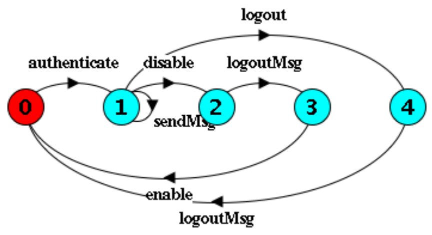

Figure 4. LTS synthesized from scenario sc.

self-loop on *logoutMsc* between 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, Fig. 1 says nothing about the possibility of the Admin disabling a User while the latter is not logged into the system (e.g., $sc_2 = \mathit{disable}, \mathit{enable}, \mathit{disable}, \mathit{enable}, \ldots$) or the possibility of the User receiving messages after he has been disabled (e.g., $sc_3 = \mathit{authenticate}, \mathit{disable}, \mathit{sendMsg}, \mathit{logoutMsg}, \ldots$). 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, Fig. 4 depicts the LTS $L(sc)$ synthesized from the Webmail scenario sc using the algorithm described in [20]. 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 requirements P of the system. LTSs 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 system 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) [12], which are capable of distinguishing possible from required behaviour.

3 Background

In this section, we review the notion of and operations over transition systems, 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 [1].

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 $L = (S, A, \Delta, s_0)$, where $S \subseteq$ States is a finite set of states, $A \subseteq Act$ is a set of labels, $\Delta \subseteq (S \times A \times S)$ is a transition relation, and $s_0 \in S$ is the initial state.*

LTSs model interaction of a (sub-)system with its environment. An example LTS is shown in Fig. 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 short for an individual transition on each action.

Modal Transition Systems (MTSs) [12], 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 2 (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 $\Delta^p \setminus \Delta^r$ as *maybe* transitions. Maybe transitions are denoted with a question mark following the label. We refer to the MTS (LTS) with a single state and an empty transition relation as the *empty* MTS (LTS). An example MTS is shown in Fig. 8(b).

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 3 (Traces) *A trace $\pi = a_0, a_1, \ldots$, where $a_i \in Act$ 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.*

We denote the set of true, maybe, possible, and false traces over a given MTS M by $\textsc{TrueTr}(M)$, $\textsc{MaybeTr}(M)$, $\textsc{PosTr}(M)$, and $\textsc{FalseTr}(M)$, respectively. For an LTS $P = (S, L, \Delta, s_0)$ we denote by $\textsc{Tr}(P)$ the set of true traces of its embedding into MTS, i.e. $\textsc{TrueTr}((S, L, \Delta, \Delta, s_0))$.

To capture the notion of elaboration of a partial description into a more comprehensive one, we use *refinement*, and say that MTSs are *equivalent* ($\equiv$) if they refine each other.

Definition 4 (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* (M, N) *is contained in some refinement relation* $R \subseteq \wp_M \times \wp_M$ *for which the following holds* $\forall \ell \in Act$:

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

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 $\mathcal{I}(M)$.

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

Definition 5 (Infinite-Trace) *An MTS* $M = (S_M, Act, \Delta^r_M, \Delta^p_M, s_{0M})$ *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. All other MTSs are called *finite-trace*. Since we intend to synthesize models from temporal logic formulas which are evaluated on infinite traces, from now on, we write "MTS" ("LTS") to mean an infinite-trace MTS (LTS), unless stated otherwise.

Parallel composition [12] captures the notion of MTSs that run asynchronously but synchronize on shared actions.

Definition 6 (Parallel Composition) *Let* M *and* N *be MTSs where* $M = (S_M, Act, \Delta^r_M, \Delta^p_M, s_{0M})$, $N = (S_N, Act, \Delta^r_N, \Delta^p_N, s_{0N})$. *Then* 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 Fig. 5.*

Merging MTSs [19] 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.

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

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

Figure 5. Rules for parallel composition.

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

Figure 6. One of the rules for the $+_{cr}$ operator.

Definition 7 (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 *and there is no MTS* $Q' \not\equiv Q$ *such that* Q' *is a CR of* M *and* N, *and* $Q' \preceq Q$.

Given two MTSs M and N that are deterministic and *consistent* (i.e., there exists an MTS that is a common refinement of both), $M +_{cr} N$ is their unique minimal common refinement.

Definition 8 (The $+_{cr}$ Operator [1]) *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 MTS. Then* $+_{cr}$ *is a symmetric operator and* $M +_{cr} N$ *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 satisfying the MT rule in Fig. 6, and the TT and MM rules of Fig. 5.*

The $+_{cr}$ differs from parallel composition only when synchronizing a maybe with a required transition. The composition of these produces a required transition instead of a maybe which parallel composition would have produced (see Fig. 5). 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.

Theorem 1 ($+_{cr}$ Builds the MCR [1]) *If* M *and* N *are deterministic MTSs, then* $M +_{cr} N$ *builds their MCR.*

4 3-valued FLTL

In our work, we assume that properties are specified using Fluent Linear Temporal Logic (FLTL) [4]. Linear temporal logics are widely used to describe behaviour requirements [4, 23, 8]. 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 [4].

In this section, we briefly describe a 3-valued variant of Fluent Linear Temporal Logic (FLTL) [4] and show that FLTL properties are preserved in all implementations of a given MTS.

FLTL [4] is a 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: $Fl = \langle I_{Fl}, T_{Fl} \rangle$ where $I_{Fl}, T_{Fl} \subseteq Act$ and $I_{Fl} \cap T_{Fl} = \emptyset$. A fluent

$$\begin{array}{llllll} \pi \models \mathit{Fl} & \triangleq & \pi^0 \models \mathit{Fl} & \pi \models \neg\phi & \triangleq & \neg(\pi \models \phi) \\ \pi \models \mathbf{X}\phi & \triangleq & \pi^1 \models \phi & \pi \models \mathbf{G}\phi & \triangleq & \forall i \geq 0 \cdot \pi^i \models \phi \end{array}$$
$$\pi \models \phi \wedge \psi \quad \triangleq \quad (\pi \models \phi) \wedge (\pi \models \psi)$$

Figure 7. Semantics of satisfaction operator.

may be initially *true* or *false* as indicated by the $\mathit{Initially}_{\mathit{Fl}}$ attribute. Every action $a \in \mathit{Act}$ induces a fluent, namely, $a = \langle a, \mathit{Act} \setminus \{a\}\rangle$. Given the set of fluents Φ, $\mathit{Fl} \in \Phi$ is an FTL formula, and other FTL formulas are defined inductively using the standard boolean connectives and temporal operators $\mathbf{X}$ (next), $\mathbf{U}$ (strong until), $\mathbf{F}$ (eventually), and $\mathbf{G}$ (always). For example, consider the property p_3 for the Webmail system described in Fig. 2. It uses fluents *logout* and *logoutMsg* derived from the actions with the same name and defined in the standard way.

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 $\pi \models \mathit{Fl}$, if and only if one of the following conditions holds:

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

In other words, a fluent holds at a time instant if and only if it holds initially, or some initiating action has occurred, and in both cases, no terminating action has yet occurred. Fig. 7 shows the satisfaction operator $\models$ for some FLTL operators [4]. In classical semantics, a formula $\phi \in$ FLTL holds in an LTS L (denoted $L \models \phi$) if $\forall \pi \in \Pi \cdot \pi \models \phi$.

The 3-valued semantics of FLTL over an MTS M returns the value of each formula $\phi \in$ FLTL in M. 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, 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. When M is an LTS, this semantics reduces to classical.

The most important property of this variant of FLTL is that refinement preserves *true* and *false* properties:

Theorem 2 (Preservation of FLTL) *Let M and N be MTSs s.t. $M \preceq N$. Then, $\forall \phi \in$ FLTL, $M \models \phi \Rightarrow N \models \phi$ and $M \not\models \phi \Rightarrow N \not\models \phi$.*

Therefore, if a property evaluates to *true* in M, it is *true* in all implementations of M, and if a property evaluates to *false* in M, it is *false* in all implementations of M. Furthermore, if a property evaluates to *maybe* in M, it is *true* in some implementations of M and *false* in others.

Finally, 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 $a \wedge \neg a$.

5 Synthesis from Properties

In this section, we describe and prove correct an algorithm for synthesizing an MTS for a safety property given

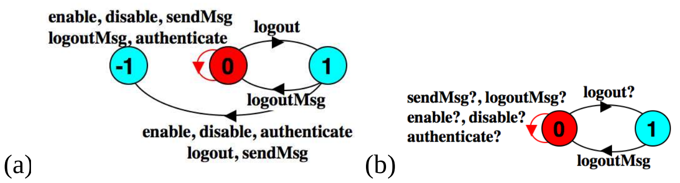

Figure 8. (a) the property LTS for p_3; (b) the MTS $M(p_3)$.

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. The algorithm is an extension of an existing algorithm for synthesizing LTSs [4].

5.1 LTS Synthesis

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 *Act* that violate ϕ and checking that the synchronous product of $B(\neg\phi)$ with L is empty [4]. $B(\neg\phi)$ 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 *Act*.

When ϕ is a safety property, $B(\neg\phi)$ has only one accepting state with 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 which corresponds to the accepting state of $B(\neg\phi)$. 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, the property LTS for p_3 of the Webmail system is shown in Fig. 8(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 [4].

The synthesis algorithm for LTSs, developed in [13], extends [4] by firstly removing all transitions not corresponding to an infinite trace and then by removing all states that are unreachable from the initial state (which always includes the error state). The resulting model is a LTS that captures all infinite traces on the system alphabet that satisfy ϕ. We denote by $L(\phi)$ the LTS generated by this procedure (e.g., $L(p_3)$ is the LTS in Fig. 8(a) with state -1 removed). By construction, $L(\phi)$ is *deterministic* and infinite-trace. Note that for an unsatisfiable property ϕ, $L(\phi)$ is the empty LTS.

5.2 MTS Synthesis

In order to overcome the limitations described in Section 2, we extend the synthesis procedure for LTSs to synthesize an MTS from a safety property ϕ, expressed in FLTL. The algorithm is called `MTSprop`:

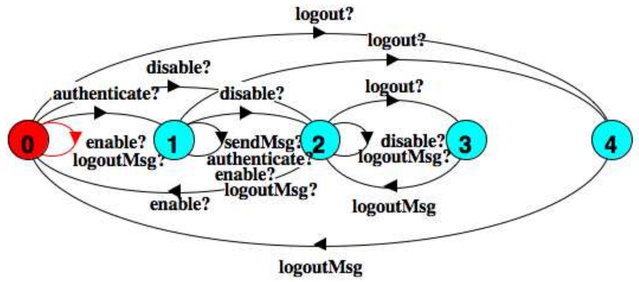

Figure 9. The MTS for property P.

1. let $L = L(\phi)$ (constructed as described in Section 5.1);
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.

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, 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`. 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 required.

For example, $M(p_3)$ is shown in Fig. 8(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 Fig. 9 (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 2, note that $M(P)$ distinguishes required from maybe *logoutMsg* transitions while $L(P)$ of Fig. 3 does not.

The MTS $M(\phi)$ constructed by `MTSprop` is not only correct, but also characterizes all MTS models that satisfy ϕ with the 3-valued interpretation of FLTL.

Theorem 3 (Characterization of ϕ) *If ϕ is a satisfiable safety property, then $\forall M \in \wp_M \cdot M \models \phi \Leftrightarrow M(\phi) \preceq M$. In particular, for all LTSs L, $L \models \phi \Leftrightarrow L \in \mathcal{I}(M(\phi))$.*

The practical implication of this 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 2 that the LTS model, $L(P)$, of the Webmail system cannot be refined to model that the trace sc_2 is required. In contrast, the MTS $M(P)$ supports this refinement by replacing the maybe transitions $0 \xrightarrow{\textit{authenticate}}_m 1$, $1 \xrightarrow{\textit{sendMsg}}_m 1$, $1 \xrightarrow{\textit{disable}}_m 2$, $2 \xrightarrow{\textit{logoutMsg}}_m 2$, with the required transitions.

6 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 [20, 10, 9], 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 conjunction 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 [7] and the synthesis algorithm presented in [20].

6.1 LTS Synthesis

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 $\textsc{Tr}(\sigma)$.

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 9 (Consistency of LTS Synthesis from Scenarios) *An LTS $L(\sigma)$ is* consistent *with a scenario specification σ if and only if $\textsc{Tr}(\sigma) \subseteq \textsc{Tr}(L(\sigma))$.*

For example, the synthesis algorithm described in [20] constructs a deterministic LTS model for each component appearing in the scenarios. Each LTS is capable of exhibiting 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 scenario sc in Fig. 1 is shown in Fig. 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.

6.2 MTS Synthesis

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 consistent with 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 $a \in Act$;

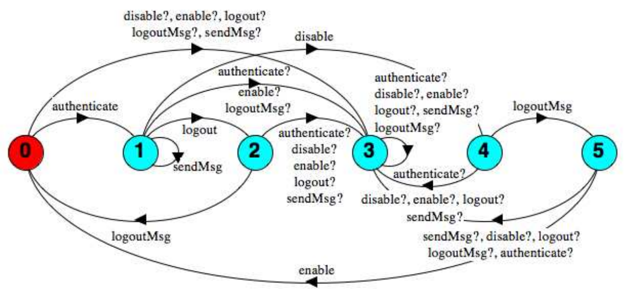

Figure 10. MTS $M(sc)$.

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 Fig. 1 is converted into $M(sc)$ of Fig. 10, where state 2 is the sink state.

It is easy to 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 4 (Correctness of MTSscen) *If σ is a scenario specification and $L(\sigma)$ is consistent with σ, then* $\text{TR}(\sigma) \subseteq \text{TRUETR}(M(\sigma))$.

More importantly, we can show that $M(\sigma)$ characterizes all models that require at least the traces of $\text{TR}(L(\sigma))$. Clearly, the degree to which the synthesized MTS characterizes the models that are consistent with σ depends on the underlying LTS synthesis algorithm. However, 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* model that preserves the scenarios can be reached by refining $M(\sigma)$.

Theorem 5 (Characterization of σ) *If σ is a scenario specification and $\text{TR}(\sigma) = \text{TR}(L(\sigma))$, then $\forall M \in \wp_M$, $M(\sigma) \preceq M$ if and only if M is consistent with σ.*

Returning to the Webmail system, the MTS of Fig. 10 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 7.

7 Synthesis From Properties and Scenarios

In this section, we discuss how to synthesize behaviour models both from safety properties and from scenarios. The synthesis process consists of merging together a model synthesized from safety properties, described in Section 5, and

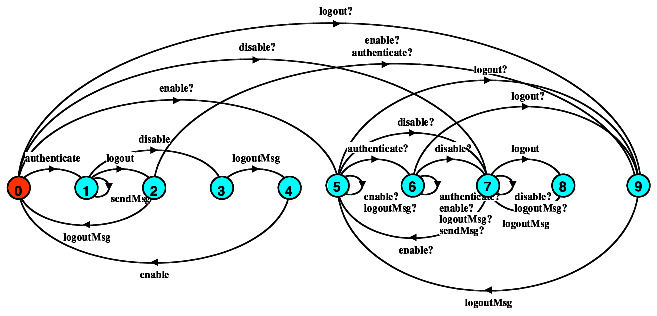

Figure 11. MTS M_{both}.

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

Figure 12. Maybe traces of $M_{both} +_{cr} M(p_4)$.

a model synthesized from scenarios, described in Section 6. Key to this process is Theorem 1: model merging preserves the required and the proscribed behaviour of the MTSs being composed. Consequently, the behaviour proscribed by properties and the behaviour required by scenarios will be preserved. 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 1 is applicable since the synthesis procedures described in Sections 5 and 6 result in deterministic models that have the same alphabet.

For our Webmail system example, the MTS $M_{both} = M(P) +_{cr} M(sc)$ is depicted in Fig. 11. This MTS captures the information provided by both scenarios and properties. Further, it can be used to reason about maybe behaviour, that is, behaviour that does not violate safety properties but has not been explored in the scenario specification. Consider the maybe trace sc_4 = *authenticate*, *logout*, *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 a missing precondition for the *authenticate* action: "A user can only be authenticated if he is not already logged in" (formalized as $p_4 = \mathbf{G}\ (\mathbf{X}\ \textit{authenticate} \Rightarrow \textit{!LoggedIn})$).

By construction, the result of merging deterministic MTSs is deterministic, and thus we can apply Theorem 1 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 maybe traces of $M_{both} +_{cr} M(p_4)$ shown in Fig. 12. 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

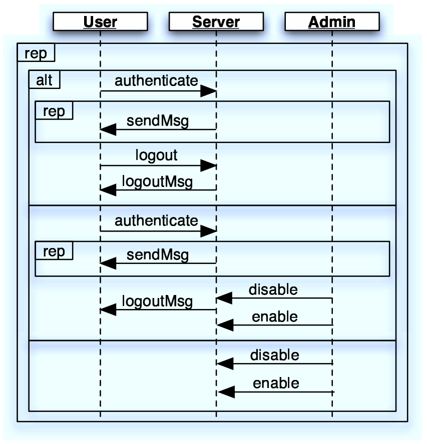

Figure 13. Extended Webmail scenarios (sc')**.**

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 system behaviour. We omit formalization of properties p_5 to p_8 due to space restrictions.

Scenarios sc_5 and sc_6 may elicit a precondition (p_5) for action *enable*: “A user can only be enabled if he was currently disabled”, 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' (Fig. 13). . 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, 7, 5, 6, ...

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), whereas sc_9 and sc_{10} may prompt missing preconditions for actions *disable* (p_7) and *authenticate* (p_8), respectively.

If a new MTS is synthesized from the existing and the newly elicited properties ($p_1 - p_8$) and the new scenario specification sc', the resulting MTS has no transitions that are possible but not required, so it is an LTS. Thus, the given scenarios and properties cover 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 require-

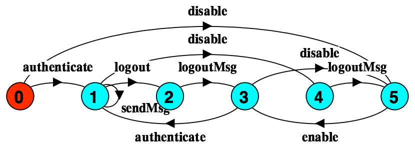

Figure 14. Final model for Webmail: $M_P +_{cr} M(p_4) +_{cr} M(p_5 \wedge \ldots \wedge p_8) +_{cr} M(sc')$**.**

ments may support behaviour model elaboration and scenario and requirements elicitation.

8 Case Study: the Mine Pump

In this section, we briefly report on a mine pump case study to which we applied the synthesis techniques described in this paper. 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 safety of the pump system.

The case study presents a number of challenges when compared to the running example used throughout the paper. Firstly, the mine pump system requires a timed model in order to capture the urgency of actions such as switching the pump off when there is methane present to avoid an explosion. Consequently, properties must make use of an explicit *tick* event, signalling the successive ticks of a global clock to which components with timed requirements synchronize. Secondly, in the running example the only component with non-trivial behaviour is the Server; 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 methane level change. Finally,the initial requirements, described informally above, are sufficiently weak, so a number of significant decisions (in terms of the problem domain) need to be made in order to elaborate the synthesized MTS.

The first MTS was synthesized from a formalization of the two key safety properties of the mine pump system: prevent flooding and prevent an explosion. Subsequently, the resulting MTS was used to validate scenarios and properties and to explore maybe behaviours. Such exploration led to elicitation of new scenarios and properties and synthesis of a more refined MTS. This process continued until the synthesis yielded an implementation – an LTS.

Validation and exploration of the synthesized models was performed by combining three different techniques: *inspection* (however, due to the number of states and transitions, this was only practical for small portions of the MTS), *animation*, which exploits the executable nature of behaviour models, and *model checking*, using a modified version of LTSA (see Section 9). Model checking can be used for elaboration as follows. When formulas constructed to query the synthesized model evaluate to maybe, they correspond to cases where the property may not hold in all refinements of the MTS. Further, the analysis yields a counterexample – a maybe trace which provides an opportunity for prompting elaboration and can be presented to stakeholders for validation. For instance, a property "The pump is on only if the water is above the high water level mark" evaluates to maybe in the model synthesized in the first iteration of the case study. A counterexample for this property highlights the fact that the pump controller may turn the pump on even when the mine sump is completely drained. This counterexample raised a number of questions about the policy for operating the pump, for example: should it only be turned on when the water is high? and once on, should it stay on until the mine has been drained?

Other interesting issues that were raised during the elaboration process were regarding assumptions on the behaviour of the pump controller environment. For example, having modeled discrete water levels, we are faced with making assumptions on how fast the water level can change. Is the sensor fast enough to catch all discrete water levels as it rises? For instance, can the sensor fail to detect water levels 4, 5, and 6 when the water rises from level 3 to 8? And if the high water mark is at level 5, how would this affect the operation of the pump controller?

In summary, at each iteration, by synthesizing operational models in the form of MTSs, we were able to reason about and explore behaviour that is between the bounds of the safety properties and the scenarios elicited up to that point. This analysis raised relevant issues and led to identifying new requirements which in turn helped produce a more refined description of the intended system behaviour.

9 Discussion and Related Work

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

On Safety Properties. In this paper, we limit our analysis just to safety properties. Instead of handling liveness properties, 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 use bounded operators, such as $\mathbf{F}_{\leq q}$, which means "eventually but in less than q time units", for capturing requirements via safety properties. This assumption is standard in requirements engineering approaches such as [22].

Alphabet Extension. In this paper, we have ignored the issue of alphabet extension, assuming instead that all properties are defined over the same alphabet, Act. In practice, fixing Act may not be 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 presented in this paper can be easily generalized to deal with alphabet extensions. Specifically, our previous study of merge handles different alphabets, as reported in [19, 1]; furthermore, we have conducted a version of the mine pump case study that included alphabet refinement.

On Tool Support. We have created prototype implementations of the synthesis algorithms and the analyses described in this paper. In particular, we have implemented the MTS synthesis algorithm which builds on existing LTS implementations developed for the LTSA tool [15], as well as algorithms for checking refinement and computing merge. We have also built on the LTSA tool for model-checking: in [1], we show that model-checking of 3-valued FLTL properties on MTSs reduces to two classical FLTL model-checking runs on LTS models and thus can be easily supported by LTSA. Ongoing work is aimed at building an integrated environment for behaviour elaboration based on MTSs.

Related Work. A number of approaches to building event-based models from properties exist [11, 21, 16, 8, 13, 14]. For instance, [11] proposed a technique for automatically translating a goal-oriented requirements model into a tabular event-based specification in the form of SCR [6]. [21, 16] developed behaviour model synthesis techniques to support animation and validation of property-based specifications. In [8], 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 [13], build *one* of the many possible event-based models that satisfy the given properties. We addressed limitations of such approaches in Section 2. An alternative, presented in [14], requires that the set of properties be strong enough to allow for a *unique* operational model that satisfies them. Our work aims at supporting elaboration so as to potentially achieve such a strong set of properties.

Operational models have also been built from scenario descriptions [20, 10, 9]. 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. [3] is related to ours in that it also consider 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) [5] augment sequence charts with the goal of describing existential and universal behaviour. We consider that there is substantial benefit in keeping universal and existential behaviour separate in the form of scenarios and properties. Synthesis approaches that produce LSC, e.g., [17], require a more expressive synthesis target (Büchi automata) but still do not support modeling and reasoning about possible yet not required system behavior. Extending LSC synthesis to modal-Büchi automata would address this.

10 Summary and Future Work

In this paper, we have presented an automated technique for constructing behavioural 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 MTS. We have shown how synthesis of MTS models supports behaviour model elaboration in addition to requirements and scenario elicitation. The approach we present integrates well with existing approaches such as goal [2, 22] and scenario-based [18] requirements engineering.

Key to success of the approach presented here is in providing adequate support for model elaboration, starting from partial models synthesized from a few scenarios and properties. To this end, we plan to further develop and implement methodologies and tools for model elaboration, to further include visualization strategies and support for elicitation and application of domain assumptions and requirements. We also intend to conduct larger case studies to continue to validate our techniques.

References

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

[2] J. Castro, M. Kolp, and J. Mylopoulos. Towards requirements-driven information systems engineering: the tropos project. *Inf. Syst.*, 27(6):365–389, 2002.

[3] C. Damas, B. Lambeau, and A. van Lamsweerde. "Scenarios, Goals, and State Machines: A Win-Win Partnership for Model Synthesis". In *Foundations on Software Engineering*, 2006.

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

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

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

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

[8] 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.

[9] K. Koskimies and E. MŁkinen. Automatic synthesis of state machines from trace diagrams. *Software Practice and Experience*, 24(7):643–658, 1994.

[10] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From mscs to statecharts. In *Distributed and Parallel Embedded Systems*. Kluwer Academic Publishers, 1999.

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

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

[13] 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.

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

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

[16] 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.

[17] J. Sun and J. S. Dong. Design synthesis from interaction and state-based specifications. *IEEE Trans. Soft. Eng.*, 32(6), 2006.

[18] A. G. Sutcliffe, N. A. Maiden, S. Minocha, and D. Manuel. Supporting scenario-based requirements engineering. *IEEE TSE*, 24(12):1072–1088, 1998.

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

[20] S. Uchitel, J. Kramer, and J. Magee. "Incremental Elaboration of Scenario-Based Specifications and Behaviour Models using Implied Scenarios". *ACM TOSEM*, 13(1), 2004.

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

[22] A. van Lamsweerde. "Goal-Oriented Requirments Engineering: From System Objectives to UML Models to Precise Software Specifications". In *ICSE'03*, 2003.

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