Existential Live Sequence Charts Revisited

German Sibay
FCEN, U. of Buenos Aires
gsibay@dc.uba.ar

Sebastian Uchitel
Imperial College London and
FCEN, U. of Buenos Aires
su2@doc.ic.ac.uk

Victor Braberman
FCEN, U. of Buenos Aires
vbraber@dc.uba.ar

ABSTRACT

Scenario-based specifications are a popular means for describing intended system behaviour. We aim to facilitate early analysis of system behaviour and the development of behaviour models in conjunction with scenarios. In this paper we define a novel scenario-based specification language with an existential semantics and that supports conditional specification of behaviour in the form of prechart and main chart. The language semantics is consistent with existing informal scenario-based and use-case based approaches to requirements engineering. The language provides a good fit with universal live sequence charts as standard existential live sequence charts do not adequately support conditional scenarios. In addition, we define a novel synthesis algorithm that, rather than building arbitrarily one of the many behaviour models that satisfy a scenario, constructs a Modal Transition System (MTS) which characterizes all behaviour models that conform to the scenario.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements / Specifications

General Terms

Design, Algorithms

Keywords

Scenarios, MTS, synthesis, partial behaviour models

1. INTRODUCTION

Operational 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, simulation and animation.

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 behaviour models have been investigated. In particular, synthesis from scenarios and use cases (e.g., [9, 16, 6, 1, 18]) has been studied extensively.

Scenario-based specifications such as Message Sequence Charts (MSCs) [7] describe how system components, the environment, and users interact in order to provide system level functionality. Their simplicity and intuitive graphical representation facilitate stakeholder involvement and make them popular for requirements elicitation. Synthesis from scenario-based specifications helps support early analysis, validation, and incremental elaboration of behaviour models.

A range of scenario description languages and associated behaviour model synthesis algorithms have been developed. Languages include features to describe alternative and repetitive behaviour, broadcast and multicast, state information, data values on messages, and symbolic instances. In addition, features that allow making explicit causality relations between different behaviours by means of conditional, triggered and preempted behaviour have been studied.

One important semantic variation among approaches is whether scenarios are interpreted as existential or universal statements. An existential scenario provides an example of system behaviour, one that the system-to-be is required to provide. A universal scenario provides a rule that all system behaviour is expected to satisfy. Although typically each approach is geared to one interpretation or the other, some languages, notably Live Sequence Charts (LSCs) [6], provide syntactic and semantic support for both interpretations. The motivation being that during the requirements process, there is a progressive shift from existential statements, in the form of examples and use-cases, to universal statements in the form of declarative properties. A scenario-based language that supports both interpretations is better equipped to support this shift.

Despite the variety of existing approaches, no language and associated synthesis algorithm suitable for describing conditional existential scenarios exists. Consider the statement "if the user inserts a valid card into the ATM, and then enters the correct password, she/he shall be able to request cash and have it dispensed by the ATM". This statement, likely to be provided in the form of a use case, is existential in that it provides an example of system execution. It also is conditional in the sense that requesting and obtaining cash is expected to be possible if the user has inserted a valid card and input the correct password.

A number of approaches [6, 9, 13] provide syntactic constructs for describing conditional or causal relations between sequences of actions. However, these take on a universal interpretation. For instance, universal LSCs (uLSCs) which describe conditional behaviour by means of a prechart and a main chart are interpreted as follows: once the prechart occurs, the main chart must occur. This is an appropriate semantics to describe statements such as "when the user's has entered an incorrect password three times in a row, the ATM must retain the user's card".

Conditional scenarios with existential semantics provide a good fit with use case based approaches. Use cases are typically interpreted existentially and are annotated with preconditions, for instance use cases for withdrawing cash, changing PIN and requiring a printed balance of accounts may all have the same precondition. These use cases are not mutually exclusive and it is expected that the system shall provide at least the three functions described by them when the precondition holds.

In addition, this semantics fits well with scenario-based elicitation methods (e.g. [17]) that adopt "what-if" questions in the form of sequences of interactions and elicit some of the system responses. Each response can be coded with a conditional existential scenario. A conditional universal scenario may be inappropriate as it may be unknown whether the response corresponds to mandatory behaviour or is simply one of the many possible system responses.

In this paper we define a novel scenario-specification language which support describing conditional existential scenarios as described above. Scenarios are described with a prechart and a main chart in the style of uLSCs, but are interpreted existentially: when the prechart has occurred, the system must be able to perform the main chart. We refer to this new kind of scenario as existential LSCs with precharts (epLSCs) to distinguish them from the existential scenarios provided in LSC which do not adequately support description of conditional existential behaviour. The correspondence between epLSCs and uLSC further supports our aim of providing a uniform framework for moving from examples to comprehensive descriptions throughout the requirements processes

In this paper we also define a behaviour model synthesis algorithm for epLSCs. The algorithm constructs modal transition systems (MTS). MTS are state-based models that can distinguish the required, possible and proscribed behaviour of the system-to-be. The MTS synthesised from an epLSC characterises, through a formal notion of refinement, all the implementations, modelled as LTS, that satisfy the epLSC.

There are various benefits to synthesising in a compact and intuitive operational representation all possible LTS of an epLSC description. Firstly, the bias of arbitrarily selecting a specific LTS is avoided. Second, the MTS can be used for analysing and exploring alternative implementations for the epLSCs. And thirdly, the MTS can be iteratively refined as new behaviour information elicited. This refinement effectively prunes the set of LTS models described by the MTS and achieves a more complete characterisation of the intended system behaviour. Iterative refinement can be prompted by traditional analysis techniques such as inspection, animation and model checking. In addition, refinement can be achieved by merging [15] the MTS with other MTS resulting from the synthesis from other scenarios (be them epLSCs or others) and declarative specifications [14].

The rest of the paper is organised as follows. We begin with background on behaviour models (Section 2) and then in Section 3 we discuss scenario-based languages and present a language for conditional existential scenarios. In Sections 4 we present an algorithm for synthesising MTSs from conditional existential scenarios. We then present a case study (Section 5), discuss our work and compare it to related approaches in Section 6 to then conclude in Section 7.

2. BACKGROUND

In this section we review behaviour models and fix notation. We start with the familiar concept of labelled transition systems (LTSs) which are widely used for modelling and analysing the behaviour of concurrent and distributed systems. An LTS is a state transition system where transitions are labelled with actions. The set of actions of an LTS is called its *communicating alphabet* and constitutes the interactions that the modelled system can have with its environment. In addition, LTSs can have transitions labelled with τ, representing actions that are not observable by the environment. An example LTS is shown in Figure 5. We use a convention that the initial state is labelled as 0. Otherwise, the numbers on states are for reference only and have no semantics. A transition labelled with several actions is shorthand for a set of transitions, one for each action.

Definition 1. (Labelled Transition System) *Let States be a universal set of states, and Act be the universal set of observable action labels and $Act_\tau = Act \cup \{\tau\}$. An* LTS *is a tuple $P = (S, A, \Delta, s_0)$, where $S \subseteq States$ is a finite set of states, $A \subseteq Act_\tau$ is the set of labels, $\Delta \subseteq (S \times A \times S)$ is a transition relation, and $s_0 \in S$ is the initial state. We use $\alpha P = A \setminus \{\tau\}$ to denote the communicating alphabet of P.*

Modal Transition Systems (MTSs) [11], 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 $(S, A, \Delta^r, \Delta^p, s_0)$, where $\Delta^r \subseteq \Delta^p$, (S, A, Δ^r, s_0) is an LTS representing* required *transitions of the system and (S, A, Δ^p, s_0) is an LTS representing* possible *(but not necessarily required) transitions. We use $\alpha M = A \setminus \{\tau\}$ to denote the communicating alphabet of M. We use δ to note the universe of MTS.*

Every LTS (S, A, Δ, s_0) can be embedded into an MTS $(S, A, \Delta, \Delta, s_0)$. Hence we shall sometimes refer to MTS with equal set of required and possible transitions as LTS. We refer to transitions in $\Delta^p \setminus \Delta^r$ as *maybe* transitions, depict them with a question mark following the label, and adopt the same conventions as for LTS regarding state numbers and initial state. An example MTS is shown in Figure 4.

Given an MTS $M = (S, A, \Delta^r, \Delta^p, s_0)$ we say M transitions on ℓ through a required transition to M', denoted $M \xrightarrow{\ell}_{\mathrm{r}} M'$, if $M' = (S, A, \Delta^r, \Delta^p, s_0')$ and $(s_0, \ell, s_0') \in \Delta^r$, and M transitions through a possible transition, denoted $M \xrightarrow{\ell}_{\mathrm{p}} M'$, if $(s_0, \ell, s'_0) \in \Delta^p$. Similarly, for $\gamma \in \{ r, p \}$ we write $M \xrightarrow{\hat{\ell}}_{\gamma} M'$ to denote that either $M \xrightarrow{\ell}_{\gamma} M'$ or $\ell = \tau$ and $M = M'$ are true, and we use $M \xRightarrow{\ell}_{\gamma} M'$ to denote $M(\xrightarrow{\tau}_{\gamma})^* \xrightarrow{\ell}_{\gamma} (\xrightarrow{\tau}_{\mathrm{r}})^* M'$. We write $M \xrightarrow{\ell}_{\gamma}$ (resp. $M \xRightarrow{\ell}_{\gamma}$) to denote that there exists an M' such that $M \xrightarrow{\ell}_{\gamma} M'$ (resp. $M \xRightarrow{\ell}_{\gamma} M'$). Finally, we use the natural extension of the above notation to words over A, for example $M \xrightarrow{w}_{\mathrm{r}} M'$ with $w = \ell_0, \ldots, \ell_n$ denotes that there exists $M_0, \ldots, M_n$ such that, $M = M_0$, $M' = M_n$, and $M_i \xrightarrow{\ell_i}_{\mathrm{r}} M_{i+1}$ for $0 \leq i < n$.

Weak refinement or simply *refinement* of MTSs captures the notion of elaboration of a partial description into a more comprehensive one, in which some knowledge over the maybe behaviour has been gained. It can be seen as being a "more defined than" relation between two partial models. An MTS N refines M if N preserves all of the required and all of the proscribed behaviours of M. Alternatively, an MTS N refines M if N can simulate the required behaviour of M, and M can simulate the possible behaviour of N.

Definition 3. (Refinement) *N is a (weak) refinement of M, written $M \preceq N$, if $\alpha M = \alpha N$ and (M, N) is contained in some refinement relation $R \subseteq \delta \times \delta$ for which the following holds for all $\ell \in Act_\tau$:*

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

LTSs that refine an MTS M are complete descriptions of the system behaviour up to the alphabet of M. We refer to them as the *implementations* of M. An MTS can be thought of as a model that represents the set of LTSs that implement it. The diversity of the set results from making different choices on maybe behaviour of the MTS.

Definition 4. (Implementation) *We say that an LTS $I = (S_I, A, \Delta_I, i_0)$ is a (weak) implementation of an MTS $M = (S_M, A, \Delta^r_M, \Delta^p_M, m_0)$, written $M \preceq I$, if $M \preceq M_I$ with $M_I = (S_I, A, \Delta_I, \Delta_I, i_0)$. We also define the set of implementations of M as $\mathcal{I}[M] = \{ I\ LTS \mid M \preceq I \}$.*

As expected, refinement preserves implementations, meaning that as an MTS is refined, the set of implementations it characterises is reduced (If $M \preceq M'$ then $\mathcal{I}[M] \supseteq \mathcal{I}[M']$).

Definition 5. (Hiding) *Let $M = (S, A, \Delta^r, \Delta^p, s_0)$ be an MTS and $X \subseteq Act$ be a set of observable actions. M with the actions of X hidden, denoted $M \backslash X$, is an MTS $(S, A \backslash X, \Delta^{r'}, \Delta^{p'}, s_0)$, where $\Delta^{\gamma'}$ with $\gamma \in \{r, p\}$ is the result of replacing all (s, ℓ, s') in Δ^γ such that $\ell \in X$ with (s, τ, s'). We use $M@X$ to denote $M \backslash (Act \backslash X)$.*

Merging MTSs [15] 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.

Given two MTS M and N, an MCR for them may not exist, in which case we say that M and N are inconsistent, or may not be unique [15]. In [4], an algorithm that builds a common refinement from consistent MTS is presented. We refer to this algorithm with the operator $+$. This algorithm is known to produce an MCR for the case when M and N have the same alphabet. For the case of different alphabets, an adaptation of the algorithm may not always produce an MCR but in these cases it produces a common refinement of M and N that approximates it.

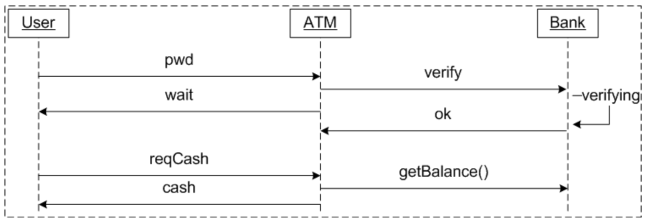

Figure 1: An existential live sequence chart (eLSC)

3. LIVE SEQUENCE CHARTS REVISITED

3.1 Sequence Charts

Sequence charts are the core of widely accepted notations for describing scenarios, notably, Message Sequence Charts (MSC) [7] and UML Interaction Diagrams. The basic syntax, depicted in Figure 1, displays vertical *lifelines* which represent component instances involved in the interaction being described. Sequence charts also depict the interactions between instance by means of arrows. These interactions, referred to as messages, can represent synchronous or asynchronous communication between component instances. In case of the former, the message represents an instantaneous event on which both instance synchronise on. In case of the latter, the message represents two instantaneous events: the event associated with the source of the arrow, the sending of the message, and the event associated with the target of the arrow. For simplicity, in this paper we shall assume that messages describe synchronous communication. The results described in this paper can be extended straightforwardly if this assumption is dropped.

Sequence charts are read from top to bottom, meaning that time is assumed to go top-down. In Figure 1, we depict a scenario in which a customer uses an ATM machine to withdraw cash. A stakeholder reading through the chart may say "The customer keys in the password and the ATM sends customer information to the bank. Then, the bank verifies the information and the ATM displays a 'please wait' message. Once the bank clears the customer, the user requests cash, the ATM gets the customer balance and dispenses the cash to the user".

A sequence chart defines a partial order of events based on the following rule: an event on a lifeline may occur if all events further above on the same lifeline have already occurred. This entails that Figure 1 does not define an order between events "verifying" and "wait"; consequently these events may occur in an arbitrary order. We refer to the set of sequences of events described by a sequence chart B as its language, written L_B. The language of the sequence chart of Figure 1 has two traces or words: *pwd*, $verify$, *wait*, $verifying$, *ok*, *reqCash*, *getBalance*, *cash* and *pwd*, $verify$, $verifying$, *wait*, *ok*, *reqCash*, *getBalance*, *cash*.

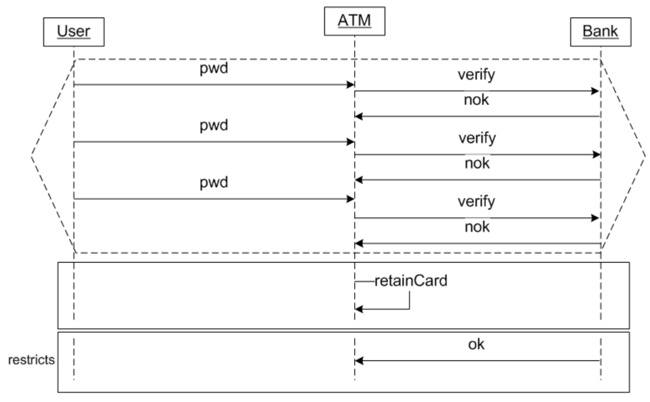

Figure 2: A universal live sequence chart (uLSC)

3.2 Live Sequence Charts - eLSC and uLSC

Many authors (e.g. [**?**, 1, 9, 16, 18] have noted the limitations of the core scenario notation described above. The key issue is the limited expressiveness of one sequence chart. Extensions have been developed to support sequence chart composition and provide control flow operations such as interleaving, repetition, concatenation, and choice. In addition, sequence charts can be annotated with state information, data values can enrich message labels, and lifelines may represent symbolic instances.

Harel et al. [6] point out that the causal relation between events remains implicit in sequence charts and that it can be beneficial to distinguish between events that trigger a scenario from the events that occur in response to the trigger. They also criticise the lack of distinction between universal and existential behaviour. Accordingly, they define an scenario-based description language based on sequence charts called Live Sequence Charts [3]. The core of LSCs, Constant LSCs [6], consists of two types of charts: Existential live sequence charts (eLSCs) and universal live sequence charts (uLSCs).

eLSCs are basic charts depicted in a dotted frame such as the one in Figure 1. We shall abstractly represent eLSCs as $\Diamond(B, \Sigma)$ where B is a basic chart and Σ is the scope of the eLSC. The scope of the the eLSC is a superset of the message labels that appear in B. We refer to the language of $E = \Diamond(B, \Sigma)$, written L_E as the set of traces that when restricted to the occurrence of events in the scope results in a sequence described by B. The intuitive semantics of an eLSC is that *there exists at least one execution* of the system-to-be that corresponds to a trace in L_E.

The purpose of including additional labels in the scope of a LSC is to restrict the occurrence of a particular message. For instance, the following sequence *pwd*, *verify*, *wait*, *verifying*, *ok*, *reqCash*, *getBalance*, *beep*, *cash*, ... is part of the language of a eLSC defined by the chart of Figure 1 with a scope that does not include *beep*, but would not be part of the language of the eLSC if *beep* were added to its scope. Syntactically, in any type of LSC, the events that are part of the scope but do not appear in the chart are depicted with an extra restricts clause as shown at the bottom of Figure 2.

uLSCs consist of two sequence charts, a prechart and a main chart where the former is depicted above the latter (see Figure 2). We represent abstractly uLSCs as $\Box(P, M, \Sigma)$ where P is the prechart and M the main chart. The semantics of a uLSC is that *in every execution* of the system-to-be, once projected onto the scope Σ, if the prechart occurs in the execution, then the main chart *must* immediately follow. Note that main chart of a uLSC is depicted in continuous frame to denote its universal nature in contrast to the dotted frame of eLSC (see Figure 2).

Consider the uLSC depicted in Figure 2, the language of its prechart contains one trace: *pwd*, *verify*, *nok*, *pwd*, *verify*, *nok*, *pwd*, *verify*, *nok* and the language of the main chart also contains one trace: *retainCard*. The scope of the uLSC is extended by the restrict clause and has the events explicitly appearing in the prechart and main chart in addition to the message in the restrict clause: {*pwd*, *verify*, *nok*, *retainCard*, *reqCash*, *getBalance*, *cash*, *ok*, *wait*, *verifying*}. An informal interpretation of the uLSC is that once a user has input the password incorrectly three times in a row, the user's card must be retained. An example of a trace that is not in the language of the uLSC is *pwd*, *verify*, *nok*, *pwd*, *verify*, *nok*, *pwd*, *verify*, *nok*, *pwd*, *verify*, *ok*, *reqCash*, ...

We now provide a formal definition of the semantics of eLSCs and uLSCs. Note that given a word w over an alphabet A, $w|_B$ with $B \subseteq A$ is the projection of w onto the alphabet B.

DEFINITION 6. (Semantics of eLSC and uLSC) *Given an infinite word $w \in Act^\omega$ we say that,*

- *w satisfies an eLSC $E = \Diamond(B, \Sigma)$, written $w \models E$, if there is a decomposition uvw' of w such that $v|_\Sigma \in L_B$.*
- *w satisfies an uLSC $U = \Box(P, M, \Sigma)$, written $w \models U$, if for every decomposition upw' of w, if $p|_\Sigma \in L_P$ then there is a decomposition mw'' of w' such that $m|_\Sigma \in L_M$.*

An LSC S defines a set of traces given by the words that satisfy the LSC: $L_S = \{w \in Act^\omega \cdot w \models S\}$. In addition, given an LTS I with a set of traces L_I we say that,

- *I satisfies E, written $I \models E$ if $L_I \cap L_E \neq \emptyset$*
- *I satisfies U, written $I \models U$ if $L_I \subseteq L_U$*

3.3 Existential LSCs with Precharts

In this section we provide a novel scenario-specification language, existential LSC with precharts which we shall refer to as epLSCs.

An epLSC consist, similarly to uLSC, of two sequence charts: a prechart and a main chart. Syntactically, epLSC differ from uLSC in its main chart appears in a dotted frame (see Figure 3). We represent abstractly an epLSCs as $\Diamond(P, M, \Sigma)$ where P is the prechart, M the main chart and Σ the scope. The intuitive semantics of an epLSC is that *in every execution* of the system-to-be projected onto the scope Σ, if the prechart occurs in the execution, then *there exists an execution from that point on* in which the main chart must follow.

Consider the epLSC depicted in Figure 3 with scope {*pwd*, *verify*, *wait*, *verifying*, *ok*, *reqCash*, *getBalance*, *cash*}, the language of its prechart contains two traces: *pwd*, *verify*, *wait*, *verifying*, *ok* and *pwd*, *verify*, *wait*, *verifying*, *ok*. The language of the main chart has only one trace *reqCash*, *getBalance*, *cash*. An informal and loose interpretation of

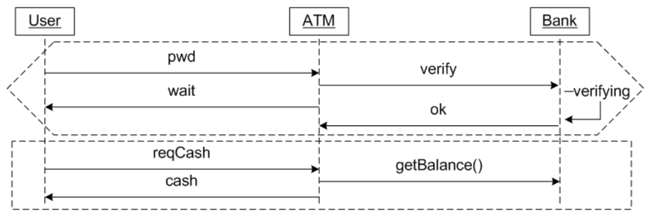

Figure 3: An existential LSC with prechart (epLSC)

the epLSC is that "once a user has input the password correctly, he or she shall be able to withdraw cash, but other behaviour could follow instead". A universal interpretation of the same scenario would yield an informal description along the lines "once a user has input the password correctly, the system shall allow the user to withdraw cash". A eLSC built from the concatenation of the prechart and main chart would state that "there exists one execution in which the user inputs correctly the password and then withdraws cash". This differs from the epLSC semantics which requires that after every correct input of a password by a user, the user must be able to withdraw cash.

Although it is possible to show a trace that conforms to this epLSC (e.g. pwd, $verify$, $wait$, $verifying$, ok, $reqCash$, $getBalance$, $cash$ or even pwd, $verify$, $wait$, $verifying$, ok, $reqCash$, $getBalance$, $beep$, $cash$), an example of a trace that does not conform to the epLSC cannot be provided. For instance, a system exhibiting pwd, $verify$, $wait$, $verifying$, ok, $requestOverdraft$, $checkCreditHistory$, $reqCash$, $\ldots$ does not necessarily violate the epLSC. The key is whether, once the prechart occurs, the system has at least one execution from that point on that satisfies the main chart, say $reqCash$, $getBalance$, $beep$, $cash$.

The interpretation of epLSCs discussed in the previous paragraphs requires defining its semantics formally in terms of a branching structure such as computation trees, or directly over an LTS, as opposed to defining the semantics over traces as is the case for uLSCs and eLSCs.

Definition 7. (Semantics of epLSC) *An LTS* $I = (S, A, \Delta, s_0)$ *satisfies an epLSC* $E = \Diamond(P, M, \Sigma)$*, written* $I \models E$ *if for all LTS* I' *such that* $I \xrightarrow{up} I'$ *and* $p|_\Sigma \in L_P$*, then for all* $m \in L_M$ *there exists a word* $m' \in A^*$ *such that* $m'|_\Sigma = m$ *and* $I' \xrightarrow{m'}$

4. MTS SYNTHESIS

In this section we define a synthesis algorithm that constructs behaviour models in the form of Modal Transition Systems (MTS) from epLSCs.

In general, the scenario synthesis problem consists in constructing a behaviour model that satisfies a given scenario description. The problem has a number of variants depending on the scenario language used, the behaviour modeling formalism chosen as a target of the synthesis, and the various additional constraints that can be imposed such as in distributed synthesis (e.g. [16]).

A stronger requirement for the synthesis is that the resulting model characterise through some notion of refinement all the behaviour models that satisfy a given scenario description. A number of techniques that perform such synthesis have been developed (e.g. [13, 14]).

Characterising in one operational model all behaviour models that satisfy a given scenario-based description is convenient as the synthesised model can then be evolved independently of the scenario description. It can be elaborated through step-wise refinement with the guarantee that the resulting, more refined, models will continue to satisfy the scenarios. Iterative refinement can be prompted by traditional analysis techniques such as inspection, animation and model checking.

Note that LTSs are not a good target for our synthesis approach as there are many of them that satisfy an epLSC. For instance the LTSs of Figures 5 and 6 satisfy the epLSC of Figure 3. A synthesis algorithm that builds one or the other would be making an arbitrary decision.

We now present an algorithm that given an epLSC E with alphabet Σ_R, produces an MTS M that characterises all LTS that satisfy the scenario, i.e. $I@\Sigma_R \in \mathcal{I}[M] \Leftrightarrow I \models E$. This entails that MTS refinement preserves the semantics of epLSCs and that MTS merge provides a composition mechanisms for epLSC scenarios. In other words, that the synthesis of an MTS from a set of epLSC can be defined as merging the MTS synthesised from each epLSCs. We first walk through an example of the algorithm's output, and then explain how the algorithm works.

Consider the MTS in Figure 4 synthesised from the epLSC of Figure 3. States 0 to 4 and state 7 of Figure 4 monitor the occurrence of all the events in the epLSC's scope. These states and their outgoing transitions guarantee that any sequence of events in the scope will lead to state 8 as soon as a trace in the language of the prechart occurs. Note that all outgoing transition from these states are maybe transitions, meaning that the intended system behaviour may or may not exhibit some of these transitions. In fact, a system that never exhibits the prechart is a valid implementation.

State 8 is reached if and only if a word in the prechart has just been recognised. From 8 there is a sequence of required transitions ($reqCash$, $getBalance$, and $cash$) leading to states 6, 5 and then 0. These transitions ensure that if the intended system behaviour where to exhibit behaviour leading to state 8, then it must provide the required transitions on $reqCash$, $getBalance$, and $cash$. In other words, these transitions guarantee the occurrence of the existential main chart. Note, however, that from states 5, 6, and 8 there is an outgoing transition for each action the epLSC's scope. This models the fact that the system-to-be does not have to satisfy the main chart in every execution that has satisfied the prechart. Furthermore, on states 5 and 6 a maybe transition on the unobservable action τ also is present. The τ transitions are needed because an LTS may have transitions beyond the scope of the epLSC leading to a point where the main chart can no longer be satisfied. Indeed, as we shall see later, a τ transition forgets the obligations contracted by a previously satisfied prechart. For the case that the prechart has just occurred, the loop on τ at state 8, ensures that there continues to exist an execution in which the main chart happens because the prechart still holds. Finally, it is important to note that the transitions from 5, 6, and 8 continue to monitor for (another) occurrence of the prechart. Hence, outgoing transitions from 5, 6, and 8 on pwd lead to state 1, while the rest go to state 0.

The MTS in Figure 4 characterises through refinement all LTS models that satisfy the epLSC of Figure 3. For example, the LTS of Figure 6, when restricted to the alphabet of

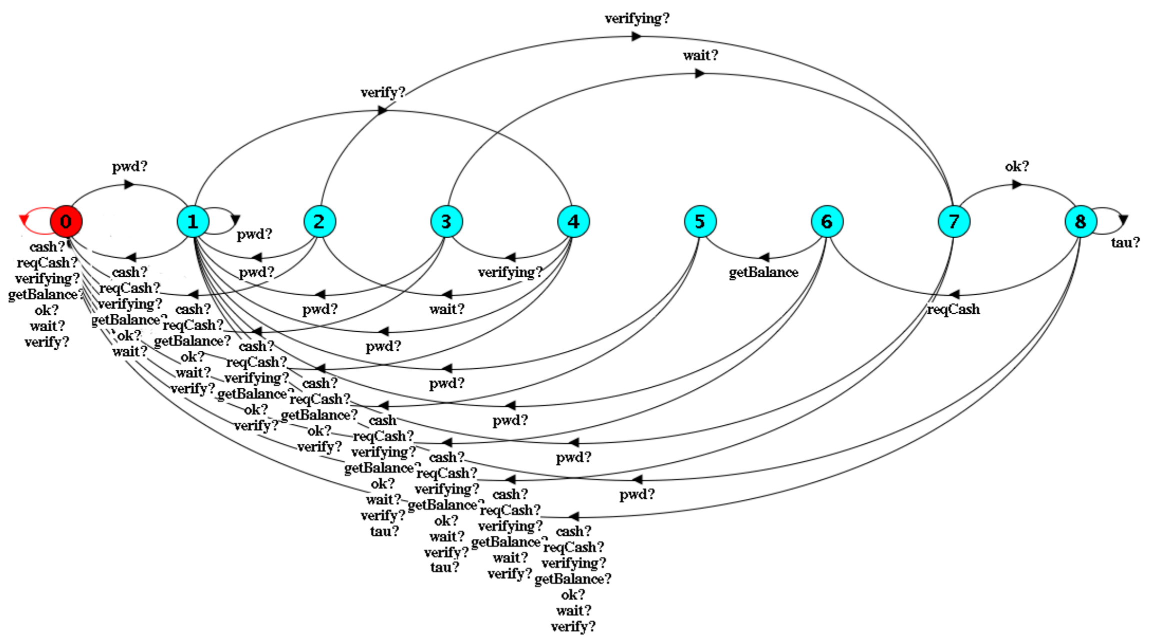
Figure 4: MTS synthesised from epLSC in Figure 3

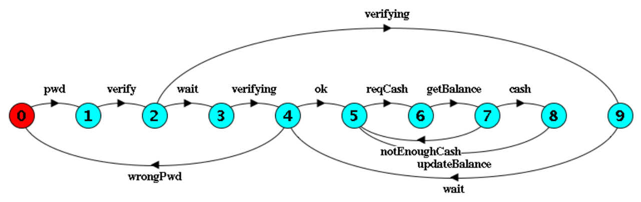
Figure 5: An implementation of MTS in Fig. 4

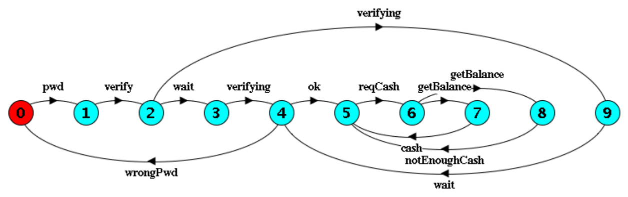
Figure 6: Another implementation of MTS in Fig. 4

the epLSC, is a refinement of the MTS of Figure 4. The refinement relation between them is $\{(0,0), (1,1), (4,2), (3,9), (2,3), (7,4), (8,5), (6,6), (5,7), (0,5), (0,6), (0,7), (0,8)\}$. It is simple to show that this implementation satisfies the epLSC of Figure 3. An interesting point to observe in this example is that the implementation has a non-deterministic choice at state 6 on label $getBalance$ which is simulated by state 6 in the MTS which does not have a non-deterministic choice on $getBalance$.

The algorithm builds an MTS with states that have the following structure $<\alpha, \Theta>$ where $\alpha \in prefixes(L_P)$ is a prefix of some word over the language of the prechart, and $\Theta \in \wp(suffixes(L_M))$ is a set of suffixes of words in the language of the main chart.

Given a state $s =<\alpha, \Theta>$, the word α describes the portion of the prechart that has been covered by the execution leading to state s. The word α is the longest suffix of the executions that lead to s that is in $prefixes(L_P)$. The set of words $\Theta \in \wp(suffixes(L_M))$ represents the remaining obligations generated by an execution leading to s. In other words, the suffixes of words in the main chart that have yet to occur in order for the activation of a prechart in an execution leading to s be satisfied.

The synthesis algorithm (see Algorithm 1) starts with an MTS with no transitions and one state (lines 3-9). The initial state of the MTS is $<\epsilon, \emptyset>$, i.e. the state in which no prefix of the epLSC has been recognised and no obligations have been acquired. The algorithm takes each unmarked state $s =<\alpha, \Theta>$ of the MTS (line 12), marks it (line 14) and then for each label in the scope, it adds transitions and possibly new states reached by these transitions (lines 15-16). Also, if the state has obligations, a τ transition is added to forget the obligations inherited from previous states. However if the prechart holds it will still hold after a τ transition and new obligations will appear (to satisfy the main chart). The criteria for adding transitions and states is based on the values of α and Θ and is encapsulated in procedure $addTransitions$. The algorithm finishes when all the states added by procedure $addTransitions$ have been marked.

The core of the algorithm is in procedure $addTransitions$ (see below). The procedure adds at least one transition on t from $s =<\alpha, \Theta>$. It may add various required transitions on t to fullfil obligations in Θ.

First, the portion of the prechart that has been recognised once t occurs is computed (line 3). In addition, the set of obligations for the state reached on t from s is computed. Obligations can emerge in two ways: as new or inherited.

If the occurrence of t completes the prechart (line 5) then all words of the main chart become new obligations (line 7), otherwise there are no new obligations (line 10).

Then the procedure adds the transitions. First by adding a non-deterministic choice over required transitions labelled t for each of the obligations of the current state that start with t. What is left from this obligations is defined by $follows(\Theta, t)$ which is the set of word θ such that $t\theta \in \Theta$. These non-deterministic choices represent the branches

```
   Data: $E = \diamond(P, M, \Sigma_R)$
   Result: $W = (S, \Sigma_R, \Delta^r, \Delta^p, s_0)$ s.t.
           $\forall I@\Sigma_R \in \mathcal{I}[W] \cdot I \models E$
1  begin
3      $\Delta^r \longleftarrow \emptyset$;
4      $\Delta^p \longleftarrow \emptyset$;
5      $s_0 \longleftarrow < \epsilon, \emptyset >$;
6      $S \longleftarrow \emptyset$;
7      $W \longleftarrow (S, \Sigma, \Delta^r, \Delta^p, s_0)$;
9      add($s_0$,$S$);
11     while unmarked($S$) $\neq \emptyset$ do
12         $< \alpha, \Theta > \longleftarrow$ get(unmarked($S$));
14         mark($< \alpha, \Theta >$);
15         foreach $t \in \Sigma_R$ do
16             addTransitions($S, < \alpha, \Theta >, t, \Delta^r, \Delta^p$)
17         if $\Theta \neq \emptyset$ then /* obligations exists          */
18
               /* $\tau$ transition that forgets them       */
20             addTransitions($S, < \alpha, \Theta >, \tau, \Delta^r, \Delta^p$)
21     return $W$;
22 end
```

Algorithm 1: Synthesis of MTS

in which the MTS shall ensure that there is at least one execution in which the obligation introduced by the main chart is exhibited (lines 12 to 23). Note that requiring a unique required transition over t to fulfil all the obligations starting with t would be too strong, as the semantics of *epLSC* does not require that it is the same branch that satisfies the common prefix of all words in the language of the main chart. If it is not the end of an obligation, what is left, is set as *inheritedObligation* (line 17). Otherwise *inheritedObligation* is the empty set as the obligation was just fulfilled. Then the obligations for the next state (line 19) will be the new obligations contracted by the last transition (L_M if the prechart holds, $\emptyset$ otherwise) and what is left from the obligation being fulfilled (if any).

In addition to required transitions over t, because of the existential nature of the semantics of *epLSC*, there may also be an execution starting with t even if it's not an obligation. Hence, in line 24 to 28 a maybe transition on t is added if there are no obligations starting with t (i.e the first symbols of Θ does not contain t). Again, it may or may not have new obligations depending on the last transition t. There are no inherited obligations here as no obligation is being fulfilled.

Note that by definition, $follows(\Theta, \tau) = \emptyset$ and $\tau \notin firsts(\Theta)$ as obligations do not contain τ. So there are no obligations by τ, they are always set as maybe transitions and so they drop inherited obligations on the current state.

The synthesis algorithm above can be proven correct and complete with respect to the semantics of epLSC. In other words, it can be proven that all implementations of the synthesised MTS from an epLSC satisfy the epLSC (Theorem 1) and that all LTS that satisfy an epLSC are implementations of the MTS synthesised from it (Theorem 2).

Theorem 1 (Correctness). *Let $E = \diamond(P, M, \Sigma)$ and W the result of applying the synthesis algorithm to E. If $I@\Sigma \in \mathcal{I}[W]$, then $I \models E$.*

The proof of this algorithm consists in assuming that there is an LTS I such that $I@\Sigma$ is an implementation of W that does not satisfy $E = \diamond(P, M, \Sigma)$ and showing that there can be no refinement relation between W and $I@\Sigma$. As I

```
1  begin
       /* get significant prefix for next state       */
3      $\alpha' \longleftarrow$ next($\alpha$, $t$);
       /* get new obligations for next state          */
5      if suffixes($\alpha'$) $\cap L_P \neq \emptyset$ then
           /* prechart holds                          */
7          $newObligations \longleftarrow L_M$;
8      else
10         $newObligations \longleftarrow \emptyset$;
           /* no new obligations                      */
       /* add transitions                             */
       /* add new required branch for each obligation
          starting with $t$                           */
12     foreach $\theta' \in$ follows($\Theta$, $t$) do
13         if $\theta' = \epsilon$ then
               /* $t$ ends with obligation            */
14             $inheritedObligation \longleftarrow \emptyset$;
15         else
               /* propagate old obligation without
                  initial $t$                         */
17             $inheritedObligation \longleftarrow \{ \theta' \}$;
19         $nextState \longleftarrow$
           $< \alpha', inheritedObligation \cup newObligations >$;
20         add(($< \alpha, \Theta >$, $t$ , $nextState$), $\Delta^p$);
21         add(($< \alpha, \Theta >$, $t$ , $nextState$), $\Delta^r$);
           /* Add new state if not previously visited
              */
23         addIfNotPresent($nextState$, $S$);
24     if $t \notin$ firsts($\Theta$) then
           /* add maybe transition                    */
26         $nextState \longleftarrow < \alpha', newObligations >$;
27         add(($< \alpha, \Theta >$, $t$ ,$nextState$), $\Delta^p$);
28         addIfNotPresent($nextState$, $S$);
29 end
```

Procedure addTransitions(S,$< \alpha, \Theta >$, t, Δ^r, Δ^p)

does not satisfy E there must be a finite execution $\Pi = \delta\beta$ over the alphabet of I such that the projection β onto the alphabet of E is in L_P and that the state reached from I on β cannot exhibit some behaviour γ required in L_W. That is $I \overset{\Pi}{\Longrightarrow} I'$ and $I' \overset{\gamma}{\not\Longrightarrow}$. The assumption that $I@\Sigma$ is a refinement of W entails that there is an W' that is refined by $I'@\Sigma$ such that $W \overset{\Pi|_\Sigma}{\Longrightarrow} W'$ and that $W' \overset{\gamma}{\not\Longrightarrow}$. From this a contradiction can be shown based on the following two properties. The first is that that any trace ending in a sequence that activates the prechart, if replayed over W using only events in the scope of E reaches a state $s =< \alpha, \Theta >$ in which the obligations imposed by the main chart are in Θ (see Property 1). The second property is that if a state s of W has an obligation w then there is a required trace from s that satisfies the main chart (see Property 2).

Property 1. *Let $W = (S, \Sigma, \Delta^r, \Delta^p, s_0)$ be the MTS resulting from the synthesis algorithm applied to $E = \diamond(P, M, \Sigma)$. For all $\beta\gamma \in Act^*$ such that $\gamma|_\Sigma \in L_P$ it is the case that $\forall s =< \alpha, \Theta > \in S \cdot (W \xrightarrow{\beta\gamma} s \Rightarrow L_M \subseteq \Theta)$*

Property 2. *Let $W = (S, \Sigma, \Delta^r, \Delta^p, s_0)$ be the MTS resulting from the synthesis algorithm applied to $E = \diamond(P, M, \Sigma)$. If $< \alpha, \Theta > \in S$, then $\forall \theta \in \Theta \cdot < \alpha, \Theta > \xrightarrow{\theta}_{\mathrm{r}}$*

The constructive proof for the completeness theorem stated below is rather long and cumbersome. Due to space limitations we omit it from this paper.

THEOREM 2 (COMPLETENESS). *Let $E = \Diamond(P, M, \Sigma)$ and W the result of applying the synthesis algorithm to E. If $I \models E$ then $I@\Sigma \in \mathcal{I}[W]$.*

5. CASE STUDY: THE MINE PUMP

In this section, we report on a case study we have conducted on a pump controller system in a mine sump [8]. In this system, a pump controller is used to prevent the water in a mine sump from passing some threshold, and hence flooding the mine. To avoid the risk of explosion, the pump may only be on when there is no methane gas present in the mine. The pump controller monitors the water and methane levels by communicating with two sensors, and controls the pump in order to guarantee the safety properties of the pump system. An epLSC exemplifying the intended execution of the mine pump controller system is depicted in Figure 7.

The case study was conducted by iterating a synthesise-analyse-elicit cycle. Firstly, an MTS is synthesised from known properties and scenarios. The scenario language and associated synthesis algorithm used are those described in this paper. The language for describing properties is the linear temporal logic of fluents for which a synthesis algorithm that builds MTS that characterise all LTS that satisfy the property has been presented in [14]. Composition of the various MTS resulting from epLSCs and FLTL properties is performed using MTS merge. In this paper, we shall only provide natural language descriptions of the properties as the formalisation of these is beyond the scope of the paper.

In the second stage, the maybe behaviour of the resulting MTS is analysed to identify missing required and proscribed behaviour. Analysis is performed using standard model-based validation techniques such as inspection (both of the textual and graphical representation of the MTS), animation, abstraction and minimisation. Tool support for these analyses is provided by the Modal Transition System Analyser (MTSA) available at:

`http://lafhis.dc.uba.ar/~suchitel/MTSA.html`

Finally, the exploration of the maybe behaviour leads to the third stage in which new scenarios and properties are elicited based on the available knowledge of the problem domain. For this particular case study existing documentation for the mine pump controller was used as a replacement of the domain expert.

The cycle was repeated until the synthesised model had no maybe transitions on events controlled by the mine pump. The final model, the LTS for the mine pump controller (see Figure 10), was obtained by then converting all transitions controlled by the environment to required, hence producing the controller that constrains as least as possible its environment.

One of the challenges of this particular case study is that the mine pump controller 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, some of the elicited properties must make use of an explicit *tick* event, signalling the successive ticks of a global clock to which components with timed requirements synchronize. This corresponds to a standard approach to modelling discrete-time in event-based formalisms [12].

5.1 First Iteration

The case study started with the epLSCs depicted in Figure 7 and 8. Note the dashed vertical line to the right of the

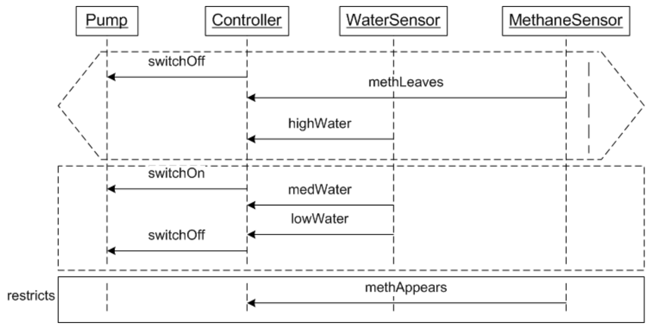

Figure 7: Scenario 1 for the Mine Pump System

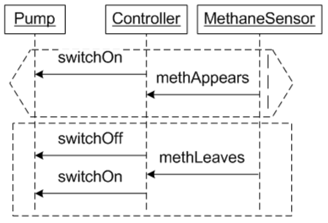

Figure 8: Scenario 2 for the Mine Pump

precharts. The line is the syntactic representation of what is called a *co-region*[7]. Co-region allow describing arbitrary interleavings of events, for instance in Scenario 1, the language of the prechart contains the 6 possible orderings of events *switchOff*, *methLeaves*, and *highWater*.

Scenario 1 exemplifies the pump being turned on when the level of the water is high and methane is not above the critical threshold and the pump is off. Scenario 2 provides an example of the the pump being switched off when the pump is on and the level of methane is above the critical threshold.

In addition, four safety properties were used. The first two correspond to the key safety properties of the mine pump system: prevent flooding (ϕ_1 - "if there is no methane and high water, the pump must on at the next state") and prevent an explosion (ϕ_2 - "if there is methane present then the pump must be off at the next state"). The other two properties (ϕ_3 and ϕ_4) simply state that the pump must not be turned on (resp. off) when it is already on (resp. off).

Note that in this case Scenarios 1 and 2 act as witnesses of the safety properties ϕ_1 and ϕ_2, they provide examples as to how the system may achieve these properties.

Having synthesised an MTS for each property and each epLSC, an MTS that combines the behaviour information of all models is produced via MTS merging.

Analysis of this first partial operational model for the mine pump controller (M_1) resulted in a number of findings. Firstly, we identified a number traces that exhibited undesired behaviour of environment controlled actions. For instance, the water level jumping from low to high without going past the medium mark. This led to producing a behaviour model M_{water} and $M_{methane}$ describing the expected behaviour of water and methane levels. These models were composed in parallel with M_1, and all subsequent refinements of M_1, to eliminate behaviours that violate assumptions on the environment.

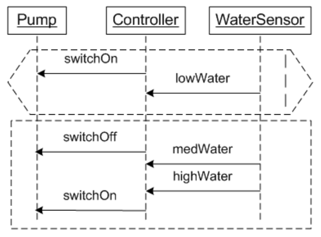

Figure 9: Scenario 3 for the Mine Pump System

5.2 Remaining Iterations

The second iteration led to the scenario of Figure 9 which provides further examples of the pump being switched off, in this case, when the level of water in the mine is low. This is to avoid the pump malfunctioning because there is no water to pump out. The example prompted in the next iteration a discussion on what is an appropriate precondition for switching the pump off. We considered three possible strategies for the pump: The pump is eager, in other words that if there is any water to pump out then the pump should remain on, or lazy, as long as there is no high water, the pump should be off, or a pump that minimises the changes in state, if it is on, it continues to be on until the water is low, and if it is off it continues to be off until water is high.

Although a number of pump strategies were explored during the case study, we report on just one of the possible evolutions of the elaboration process. In this paper we opt for an variation of an eager pump as captured by property ϕ_5 stating that water low is the precondition for switching pump off and property ϕ_6 which described the precondition for switching the pump on: the pump can be turned on only if water is not low and methane is not present. Finally, through inspection, we identified two maybe transitions, one for switching on and the other for switching off the pump, that were removed and a third maybe transition for switching the pump on that was converted to a required transition.

Having converged to an MTS in which the only actions with maybe transitions were those controlled by the environment ({ *methAppears*, *methLeaves*, *medWater*, *highWater*, *tick*, *lowWater*, τ }), the iterative synthesise-analyse-elicit cycle concluded. The resulting model captures the intended behaviour of the pump controller while leaving open assumptions on how the environment will behave. By converting all maybe transitions to required, a model for the pump controller that makes as least possible assumptions on the environment is obtained. The resulting LTS is depicted in Figure 10.

6. DISCUSSION AND RELATED WORK

A variety of scenario-based notations with diverse features and semantics have been developed. We focus our discussion on those with features that relate to the precharts of epLSC. The use of precharts or triggers to augment the expressiveness of sequence charts notations has been investigated by several authors. However, to the best of our knowledge, all approaches adopt a universal semantics. Krüger [9] extends MSC with triggers associating a universal semantics to them ("if a certain interaction pattern has occurred in the system, then another one is inevitable"). Sengupta and Cleaveland [13] also present a triggering mechanism with universal interpretation but triggers are specified component-wise rather than system-wide. In the original formulation of LSCs [3], Damm and Harel introduce precharts for both existential and universal LSCs. However, the semantics of an existential LSC with a prechart P and main chart M is equivalent to that of an existential LSC with a main chart PM and no prechart. Hence, in this case the prechart in existential LSCs results in a formatting option rather than a semantically meaningful element. In fact, in later developments of LSCs (e.g. [1, 10]) the prechart for existential LSCs is dropped.

The semantics of epLSC can be understood as a fragment of the temporal branching logic CTL*. Informally, they stand for a formula of the formula $AG(p \rightarrow E\ m)$ where p and m codify the language of the prechart and main chart. Indeed, the semantics cannot be formulated in terms of the linear temporal logic LTL, traces or histories as the semantics of uLSC and eLSC [1] or the triggered MSC in [9] can.

Many of the approaches to scenario-based specification provide synthesis algorithms that produce operational behaviour models. As discussed previously the result of synthesis can be one of the many possible behaviour models that satisfy the scenario description or a behaviour model that characterises through some notion of refinement all the behaviour models that satisfy a given scenario description.

Given a scenario description interpreted existentially, it is possible to synthesise a behaviour model M that represents the lower bound to the expected system behaviour, i.e. M "does as least as possible" while still providing the existential scenarios. This model characterises via trace inclusion or simulation all behaviour models that satisfy the scenarios: If N can simulate or includes the traces of M, then it satisfies the scenario description. Approaches such as [16, 18] provide synthesis algorithms of this kind.

Alternatively, given universal scenarios, it is possible to synthesise a model M that does "as much as possible" while preserving the scenarios. This model provides an upper-bound to the intended system behaviour and can also be thought of as characterising all behaviour models that satisfy the scenarios: If N is simulated by M, then N satisfies the universal scenario description. Approaches such as [1] when restricted to uLSC and [13] provide this style of synthesis.

In [14] we show that traditional, two valued, behaviour models such as LTS or statecharts cannot adequately model descriptions that contain both existential and universal statements such as in a combination of eLSCs and uLSCs. In other words, that it is not possible to build an LTS, for example, that characterises all LTS that satisfy the mixed modality scenario description. Roughly, this is because refinement notions for traditional behaviour models can interpret the model as an upper-bound or lower-bound to the expected behaviour of the system but cannot support both bounds simultaneously. Consequently, approaches to synthesis that support combinations of existential and universal scenarios are limited to providing an example of a behaviour model that satisfies the scenario description. This is the case for algorithms that synthesise behaviour models from uLSC and eLSC such as those given in [2] and [5]. This paper differs from [14] in that the MTS synthesis method proposed in [14] works only for existential scenario languages without precharts.

In this paper, a three valued behaviour model is used as the target for synthesis. This step up in expressiveness is what allows the definition of a synthesis algorithm that

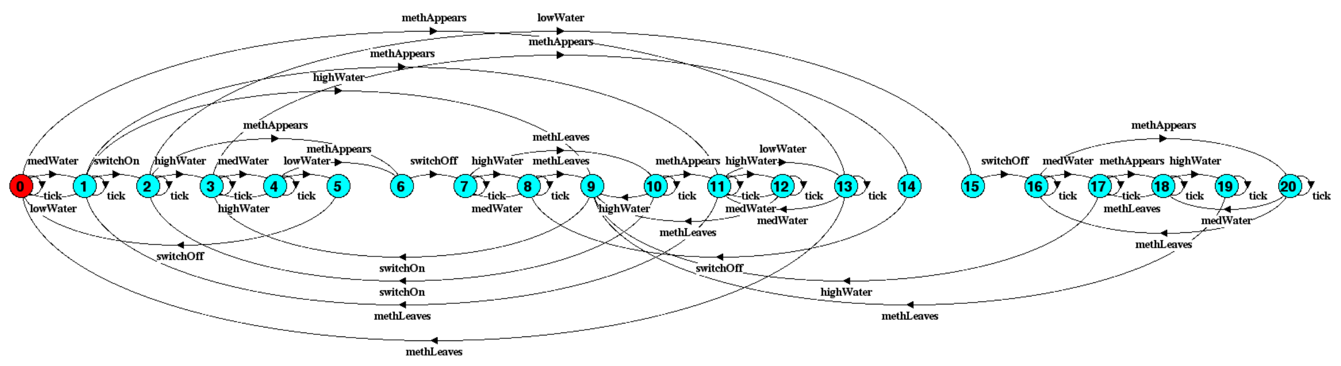

Figure 10: Final MTS

characterises all LTS models that satisfy an epLSC. Such a result cannot be achieved by synthesising two-valued behaviour models such as LTS due to the fact that both upper and lower bound behaviour must be described: If the system were to perform the prechart (upper-bound, as the system is not required to) then the system must be able to perform the main chart (lower-bound, the system is required) but may be able to exhibit other behaviour (upper-bound, it is not required to do so).

7. CONCLUSIONS

In this paper we have defined a scenario language with existential semantics and that supports conditional specification of behaviour. The semantics is in line with scenario-based and use-case based approaches to requirements engineering and provides a better fit with uLSCs as the standard eLSCs do not adequately support precharts. This correspondence between epLSC and uLSC further supports our aim of providing a uniform framework for moving from examples to comprehensive descriptions throughout the requirements process. In addition, we have defined a novel synthesis algorithm that constructs MTS whose elaboration guarantees the preservation of scenarios.

In future work, we aim to develop an MTS synthesis algorithm for uLSCs and more generally, to continue to develop support for elicitation and elaboration of behaviour models. We also intend to conduct larger case studies to continue to validate our techniques.

8. ACKNOWLEDGMENTS

The work reported herein was partially supported by CONICET, the Leverhulme Trust, EPSRC grant PBM C541133/1, UBACyT X020, and ANPCyT PICT grants 32440 and 11738.

9. REFERENCES

[1] Y. Bontemps, P. Heymans, and P.-Y. Schobbens. From live sequence charts to state machines and back: A guided tour. *IEEE TSE*, 31(12):999–1014, 2005.

[2] Y. Bontemps, P.-Y. Schobbens, and C. Löding. Synthesis of open reactive systems from scenario-based specifications. *Fundam. Inform.*, 62(2):139–169, 2004.

[3] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. In *FMOODS*, vol. 139 of *IFIP Conference Proceedings*, 1999.

[4] D. Fischbein and S. Uchitel. "On Consistency and Merge of MTS". In *International Workshop on Living with Uncertainty, ASE'07*, 2007.

[5] D. Harel and H. Kugler. Synthesizing state-based object systems from LSC specifications. *Int. J. on Foundations on Computer Science*, 13(1):5–51, 2002.

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

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

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

[9] I. Kruger. *"Distributed system design with message sequence charts"*. PhD thesis, Technical University of Munich, 2000.

[10] H. Kugler, M. J. Stern, and E. J. A. Hubbard. Testing scenario-based models. In *Fundamental Approaches to Software Engineering*, volume 4422 of *LNCS*, pages 306–320. Springer, 2007.

[11] K. Larsen and B. Thomsen. "A Modal Process Logic". In *Logic in Computer Science*, pages 203–210, 1988.

[12] A. W. Roscoe, editor. *A classical mind: essays in honour of C. A. R. Hoare.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994.

[13] B. Sengupta and R. Cleaveland. Triggered message sequence charts. *IEEE TSE*, 32(8):587–607, 2006.

[14] S. Uchitel, G. Brunet, and M. Chechik. Behaviour model synthesis from properties and scenarios. *International Conference on Software Engineering*, pp. 34–43, 2007.

[15] S. Uchitel and M. Chechik. "Merging Partial Behavioural Models". In *Foundations of Software Engineering*, pages 43–52, 2004.

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

[17] K. Zachos, N. Maiden, and A. Tosar. Rich-media scenarios for discovering requirements. *IEEE Software*, 22(5):89–97, 2005.

[18] T. Ziadi, L. Helouet, and J.-M. Jezequel. Revisiting statechart synthesis with an algebraic approach. In *International Conference on Software Engineering*, pp. 242–251, 2004.