Project no.:	**PIRSES-GA-2011-295261**
Project full title:	**Mobility between Europe and Argentina applying Logics to Systems**
Project Acronym:	**MEALS**
Deliverable no.:	**5.1 / 1**
Title of Deliverable:	**Synthesis of Modal Transition Systems from Triggered Scenarios**

Contractual Date of Delivery to the CEC:	**1-Apr-2013**
Actual Date of Delivery to the CEC:	**15-Mar-2013**
Organisation name of lead contractor for this deliverable:	**IMP**
Author(s):	**German Emir Sibay, Victor Braberman, Sebastian Uchitel, Jeff Kramer**
Participants(s):	**UBA, IMP, ULEIC, UNR, ITBA**
Work package contributing to the deliverable:	**WP5**
Nature:	**R**
Dissemination Level:	**Public**
Total number of pages:	**62**
Start date of project:	1 Oct. 2011 **Duration:** 48 month

Abstract:

Synthesis of operational behaviour models from scenario-based specifications has been extensively studied. Focus has been mainly on either existential or universal interpretations. One noteworthy exception is Live Sequence Charts which provides expressive constructs for conditional universal scenarios and some limited support for non-conditional existential scenarios. In this paper we propose a scenario-based language that supports both existential and universal interpretations for conditional scenarios.
Existing model synthesis techniques use traditional two-valued behaviour models, such as Labelled Transition Systems. These are not sufficiently expressive to accommodate specification languages with both existential and universal scenarios. We therefore shift the target of synthesis to Modal Transition Systems, an extension of Labelled Transition Systems that can distinguish between required, unknown and proscribed behaviour to capture the semantics of existential and universal scenarios. Modal Transition Systems support elaboration of behaviour models through refinement, which complements an incremental elicitation process suitable for specifying behaviour with scenario-based notations. The synthesis algorithm that we define constructs a Modal Transition System that uses refinement to characterise all the Labelled Transition Systems models that satisfy a mixed, conditional existential and universal scenario-based specification. We show how this combination of scenario language, synthesis and Modal Transition Systems supports behaviour model elaboration.

Note:

This deliverable is based on material that has been accepted for publication in IEEE Trans. on Software Engineering.

This project has received funding from the European Union Seventh Framework Programme (FP7 2007-2013) under Grant Agreement Nr. 295261.

Contents

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. These models provide a basis for a wide range of automated (and semi-automatic) analysis techniques, such as model-checking, simulation and animation.

One of the limitations of operational behaviour modelling is the complexity of building the models in the first place. Operational 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 operational behaviour models has been investigated. In particular, synthesis from scenarios and use cases has been studied extensively ([1, 2, 3, 4, 5]).

Scenario-based specifications such as Message Sequence Charts (MSCs) [6] 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 making them popular for requirements elicitation. Model synthesis from scenario-based specifications facilitates early analysis, validation, and incremental elaboration of behaviour models.

A range of scenario description languages and associated behaviour model synthesis algorithms have been developed (e.g., [1, 7, 8]). Although they differ in many aspects, a noteworthy semantic distinction 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 each approach is typically geared to one interpretation or the other, some languages, notably Live Sequence Charts (LSCs) [3], provide syntactic and semantic support for both interpretations. The motivation is 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.

1.1 Existential Triggered Scenarios

Despite the variety of existing approaches, no language and associated synthesis algorithm is suitable for describing conditional existential scenarios. 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 is existential in that it provides an example of system execution. It is also 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 [3, 1, 7] provide syntactic constructs for describing conditional or causal relations between sequences of actions. However, these take 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 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, as would be required by universal interpretation, and it is expected that the system shall provide at least that functionality when the precondition holds.

In addition, this semantics fits well with scenario-based elicitation methods (e.g. [9]) that adopt "what-if" questions in the form of sequences of interactions that elicit system responses. Each response can be codified with a conditional existential scenario, as opposed to a conditional universal scenario, as it is often unknown if the system response is mandatory or simply one of the many possible system responses.

1.2 The Synthesis Problem

A current limitation of approaches that synthesise operational models from scenario-based specifications is that the synthesised operational models, such as labeled transition systems (LTSs) [10], are typically assumed to be complete descriptions of the system behaviour; that is, that they classify every behaviour as either being required or prohibited in the system-to-be. The required behaviour is described by the transitions that appear in the operational model. The proscribed behaviour is defined as anything that is not described by the model's transitions. This completeness assumption is problematic if these behaviour models are to be built from scenario based-specifications which are inherently partial.

Traditional refinement notions such as trace inclusion or *simulation* [11] can overcome this limitation to some extent allowing an operational model to represent an upper or lower bound on the intended system behaviour. For instance an LTS may be interpreted as describing the safe behaviour of the system and any system that exhibits less behaviour, or less non-deterministic behaviour is acceptable. Alternatively an LTS may be interpreted as partially describing the behaviour of the system-to-be and any system that exhibits more behaviour is acceptable.

The problem is that if behaviour models are to be synthesised from rich scenario based languages that combine existential and universal scenarios, as first envisioned in [3], the target synthesis formalism cannot be in the form of traditional behaviour models such as LTS. These are not capable of capturing simultaneously both the upper and lower bounds that universal and existential statements provide.

1.3 MTS Models as Synthesis Target

Partial behaviour models, such as Modal Transition Systems (MTS) [12], distinguish between three kinds of behaviour, required, proscribed and unknown. MTS can therefore describe *both* an upper and a lower bound to the intended system behaviour, allowing both bounds to be refined simultaneously. MTS are equipped with two kinds of transitions: *required* transitions and *possible* transitions. The former provide a lower bound to system behaviour, while the latter provide the upper bound.

The semantics of a partial behaviour model can be thought of as a set of traditional behaviour models. For instance, MTS semantics can be given in terms of sets of LTSs that provide all of the behaviour required by the MTS, do not provide any of the behaviour proscribed by the MTS, and make arbitrary decisions on the MTS's unknown behaviour. Intuitively, as more information becomes available, unknown or unclassified behaviour is transformed into either required or proscribed behaviour. The notion of refinement [13] between MTSs captures this intuition formally and provides an elegant way of describing the process of behaviour model elaboration as one in which behaviour information is acquired and introduced into the behaviour model incrementally, gradually refining an MTS until it characterises a single LTS.

MTSs have been studied extensively, and a number of theoretical results and practical algorithms to support reasoning and elaboration of partial behaviour models expressed in this formalism have been published [14, 12, 13, 15, 16, 17]. In particular, it has been shown that MTSs (e.g. [17]) can support behaviour model elaboration when used as the target of synthesis approaches because the result of the synthesis is a model that characterises all LTSs that satisfy the source specification.

Capturing all behaviour models that comply with a scenario description in an operational representation has a number of advantages: *i)* the bias of arbitrarily selecting one of the many behaviour models that satisfy the scenario description is avoided; *ii)* the partial behaviour model can be used for analysing and exploring alternative implementations for the scenarios; *iii)* the partial behaviour model can be iteratively refined as new behaviour information is elicited.

1.4 Paper Contribution and Outline

In this paper *we define a scenario-specification language which includes support for describing both conditional existential and conditional universal scenarios*. Scenarios are described with a trigger and a main chart in the style of uLSCs. However, they can be interpreted existentially: when the trigger has occurred, the system should be able to perform the main chart, hence existential triggered scenarios (eTSs). We distinguish them from the existential and universal (and which are catered for in this approach too) scenarios provided in LSC which do not adequately support description of conditional existential behaviour. These triggered scenarios also *support state-based conditions* for triggers that greatly simplify the specification of triggering conditions.

In addition, *we define a behaviour model synthesis algorithm* for existentially and universally triggered scenarios. The algorithm constructs a modal transition system (MTS) that characterises via refinement all LTS models that conform to both existential and universal triggered scenarios.

Finally, *we show how iterative and incremental behaviour model elaboration can be supported*. By providing both existential and universal forms of triggered scenarios we aim to better support the vision of a uniform framework for moving from examples to comprehensive descriptions throughout the requirements process. We support triggered scenarios and MTS synthesis in conjunction with other existing MTS synthesis and analysis techniques such as merging [18], refinement [12], synthesis from temporal logic [17], model checking [19], inspection and animation [20].

The rest of the paper is organised as follows. We begin with background on behaviour models (Section 2) and then (Section 3) discuss scenario-based languages and present a language for

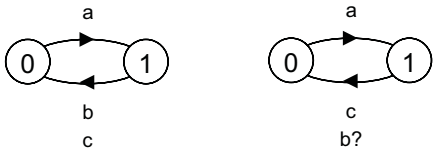


Figure 1: An LTS (on the left) and an MTS (on the right)

conditional existential and universal scenarios. In Section 4 we present an algorithm for synthesising MTSs from conditional existential scenarios which we use in the presentation of a case study (Section 5). We discuss our work and compare it to related approaches in Section 6, and conclude in Section 7.

2 Background

2.1 Transition Systems

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 [21]. 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 on the left in Figure 1. 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 several transitions, each labelled by one of the actions.

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 $\alpha P = A \backslash \{\tau\}$ to denote the communicating alphabet of P. We use $\mathcal{LTS}$ to denote the set of all LTSs.

Modal Transition Systems (MTSs) [12], allow for explicit modelling of what is *not* known, extending 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* behaviour of the system and (S, A, Δ^p, s_0) is an LTS representing *possible* (but not necessarily required) behaviour. We use $\alpha M = A \setminus \{\tau\}$ to denote the communicating alphabet of M.

Every LTS (S, A, Δ, s_0) can be embedded into an MTS $(S, A, \Delta, \Delta, s_0)$. Hence we sometimes refer to MTSs in which the set of possible transitions and the set of required transitions are

identical as LTSs. 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 on the right of Figure 1.

It is sometimes useful to hide selected transitions from a model to reduce visible complexity.

Definition 3. *(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 $(S, A\backslash X \cup \{\tau\}, \Delta^{r\prime}, \Delta^{p\prime}, s_0)$, where $\Delta^{\gamma'}$ with $\gamma \in \{r, p\}$ is the result of replacing all (s, ℓ, s') in Δ^γ, where $\ell \in X$, with (s, τ, s'). We use $M@X$ to denote $M\backslash(Act\backslash X)$.

Given an MTS $M = (S, A, \Delta^r, \Delta^p, s_0)$ we say M becomes M' via a required transition labelled by ℓ, 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 that M becomes M' via a possible transition labelled by ℓ, denoted $M \xrightarrow{\ell}_{\mathrm{p}} M'$, if $(s_0, \ell, s_0') \in \Delta^p$. Similarly, for $\gamma \in \{\mathrm{r}, \mathrm{p}\}$ we write $M \xrightarrow{\hat{\ell}}_\gamma M'$ to denote that either $M \xrightarrow{\ell}_\gamma M'$ or that $\ell = \tau$ and $M = M'$. We use $M \overset{\ell}{\Longrightarrow}_\gamma M'$ to denote $M(\xrightarrow{\tau}_\gamma)^* \xrightarrow{\ell}_\gamma (\xrightarrow{\tau}_\gamma)^* M'$.

Let $w = w_1 \dots w_k$ be a word over Act_τ. Then $M \xrightarrow{w}_\gamma M'$ means that there exist $M_0, \dots, M_k$ such that $M = M_0$, $M' = M_k$, and $M_i \xrightarrow{w_{i+1}}_\gamma M_{i+1}$ for $0 \le i < k$. For a finite w we write $M \xrightarrow{w}_\gamma$ to mean $\exists M' \cdot M \xrightarrow{w} M'$. If $w = w_1 \dots w_k \dots$ is an infinite word over Act_τ then $M \xrightarrow{w}_\gamma$ means there exist $M_0, \dots, M_k, \dots$ such that $M = M_0$ and $M_i \xrightarrow{w_{i+1}}_\gamma M_{i+1}$ for every i. We extend $\Longrightarrow_\gamma$ to words in the same way as we do for $\longrightarrow_\gamma$. We say that w can be replayed over M or that w is a (finite or infinite depending on w) trace of M if $M \xrightarrow{w}_{\mathrm{p}}$.

Let $s \in S$ then we note M_s the MTS obtained by setting the initial state of M to s. Formally, if $M = (S, A, \Delta^r, \Delta^p, s_0)$ then $M_s = (S, A, \Delta^r, \Delta^p, s)$. Finally, we use $\mathcal{MTS}$ to denote the set of all MTSs.

Weak alphabet refinement [18], or simply *refinement*, of MTSs captures the notion of elaboration of a partial description into a more comprehensive one, in which some knowledge of 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 4. *(Refinement)* Let MTSs N and M such that $\alpha M \subseteq \alpha N$. N is a *weak alphabet refinement* of M, written $M \preceq N$, if $(M, N@\alpha M)$ is contained in some *weak alphabet refinement relation* $R \subseteq \mathcal{MTS} \times \mathcal{MTS}$, for which the following holds for all $\ell \in Act_\tau$ and for all $(M', N') \in R$:

$$1.\quad \forall \ell \cdot \forall M'' \cdot \quad (M' \xrightarrow{\ell}_{\mathrm{r}} M'' \Longrightarrow \exists N'' \cdot N' \overset{\hat{\ell}}{\Longrightarrow}_{\mathrm{r}} N'' \ \wedge \ (M'', N'') \in R)$$

$$2.\quad \forall \ell \cdot \forall N'' \cdot \quad (N' \xrightarrow{\ell}_{\mathrm{p}} N'' \Longrightarrow \exists M'' \cdot M' \overset{\hat{\ell}}{\Longrightarrow}_{\mathrm{p}} M'' \ \wedge \ (M'', N'') \in R)$$

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 the maybe behaviour of the MTS.

Definition 5. *(Implementation)* We say that an LTS $I = (S_I, A, \Delta_I, i_0)$ is an implementation of an MTS $M = (S_M, A, \Delta_M^r, \Delta_M^p, m_0)$, 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 $\mathcal{I}[M] = \{I \in \mathcal{LTS} \mid M \preceq I \ \wedge \ \alpha M = \alpha I\}$.

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 $\mathcal{I}[M] \supseteq \mathcal{I}[M']$).

Merging MTSs ([22, 18]) is the process of combining what is known from each partial behaviour description; in other words, it is the construction of the least possible refined MTS that includes all the required and all the prohibited behaviours from each MTS. Formally, merging two MTSs is related to finding their common refinements.

Definition 6. *(Common Refinement)* We say that an MTS C is a common refinement of MTSs M and N if $M \preceq C$ and $N \preceq C$. We say that C is a minimal common refinement (MCR) of M and N if for all common refinements C' of M and N, $C' \preceq C$ implies $C \preceq C'$.

Given two MTS, if no common refinement exists we say that they are inconsistent. Two consistent MTS may have one, many or no minimal common refinements (MCR). Depending on the case, merging two MTS corresponds respectively to constructing the unique MCR (this model describes exactly all the common implementations of the models being merged), selecting one of the multiple MCRs or selecting an MCR up to some bound in the state space. Note that if a unique minimal common refinement exists, merge amounts to conjunction. In [18], practical algorithms for supporting merge are defined. We refer to the process of merging with the operator + and assume that when multiple MCRs exist the operator arbitrarily returns one of them. In order to characterise the intersection in general, a slightly more expressive formalism, Disjunctive MTS, is needed. For simplicity we limit the scope of this paper to MTS.

The semantics of the triggered scenarios language presented in this paper is defined over computation trees. A computation tree is an LTS in which every non-initial state has a unique parent.

Definition 7. *(Computation Tree)* A computation tree (S, A, Δ, s_0) is an LTS in which if $(x, a, y) \in \Delta$ and $(x', a', y) \in \Delta$ then $x = x'$ and $a = a'$. The computation tree T of an LTS L is an LTS resulting from unwinding [23] L from its initial state. We refer to a branch of a tree as a sequence, infinite or finite, of transitions $b = (x_0, a_1, x_1) \dots (x_j, a_j, x_{j+1}) \dots$ with x_i states of T and a_i in the alphabet of T. In addition we say that b starts at x_0. If b is finite then $b = (x_0, a_1, x_1) \dots (x_n, a_n, x_{n+1})$ and we say that it ends at x_{n+1}. Note that, in both cases, x_0 is not necessarily the initial state. A branch is *complete* if the branch is infinite or, if it is finite, its ending state has no outgoing transition. Finally, we refer to the sequence of labels along a branch as the word defined by that branch.

2.2 Sequence Charts

Sequence charts are the core of widely accepted notations for describing scenarios, notably, Message Sequence Charts (MSC) [6], UML Interaction Diagrams and Live Sequence Charts [3]. The basic syntax, depicted in Figure 2, displays vertical *lifelines* which represent component

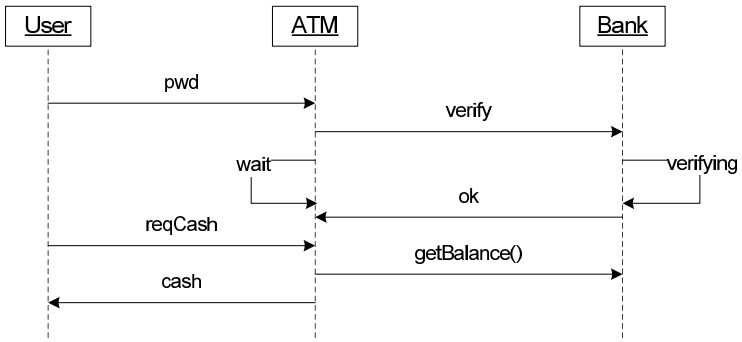


Figure 2: A MSC of an ATM

instances involved in the interaction being described. Sequence charts depict the interactions between instances by means of arrows. These interactions, referred to as messages, can represent synchronous or asynchronous communication between component instances. In the former case, the message represents an instantaneous event on which both instances synchronise. In the latter case, the message represents two instantaneous events: the sending event associated with the source of the arrow, and the receiving event associated with the target of the arrow. For simplicity, in this paper we shall assume that messages describe synchronous communication and that arrows cannot cross each other.

Sequence charts are read from top to bottom, meaning that time is assumed to go top-down. In Figure 2, 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". Note that a scenario abstracts from some of the detail, focusing on a particular aspect of the system being modeled. In Figure 2 it is not specified how the ATM interacts with the user before allowing cash withdrawal. It could be through a series of menu options or in a single step; however we are only interested in the fact that after logging in the user can withdraw cash.

Sequence charts are abstractly represented as Labelled Partial Orders (LPO). This is a standard way of giving semantics to MSC or UML Interaction Diagram [6].

Definition 8 (Labelled Partial Order (LPO))**.** A Labelled Partial Order is a tuple $\langle L, \leq, \lambda, \Sigma \rangle$ where

- L is a finite set of *locations*
- $\leq \subseteq L \times L$ is a partial order relation over L that is reflexive (i.e., $l \leq l$), anti-symmetric (i.e., $l \leq l', l' \leq l \Longrightarrow l = l'$) and transitive (i.e., $l \leq l', l' \leq l'' \Longrightarrow l \leq l''$).
- $\lambda : L \to \Sigma$ is a labeling function.

As we are assuming synchronous communication a location is just a message (otherwise we would have to consider the origin and the target of a message as two different locations). Let G

be an LPO. We define $|G|$ as the number of locations in G. An example of an LPO is $G = \langle L, \leq, \lambda, \Sigma \rangle$ where $L = \{m_1, \ldots, m_8\}$, $\leq$ is the reflexive and transitive closure of $\{(m_1, m_2), (m_2, m_3), (m_2, m_4), (m_3, m_5), (m_4, m_5), (m_5, m_6), (m_6, m_7), (m_7, m_8)\}$, and $\lambda = \{(m_1, pwd), (m_2, verify), (m_3, verifying), (m_4, wait), (m_5, ok), (m_6, reqCash), (m_7, getBalance()), (m_8, cash)\}$.

To relate a scenario with the system's behaviour we have to be able to associate an LPO with a sequence of actions (i.e. message labels).

Definition 9 (Linearisation)**.** A linearisation of an LPO $\langle L, \leq, \lambda, \Sigma \rangle$ is a word $u = e_0 \ldots e_n \in \Sigma^*$ such that the LPO $\langle \{0, \ldots, n\}, \leq_{\mathbb{N}}, \lambda_u, \Sigma \rangle$ is isomorphic to $\langle L, \leq', \lambda, \Sigma \rangle$ for some total order $\leq' \supseteq \leq$, and

- $\leq_{\mathbb{N}}$ is the order of the natural numbers
- the labeling function maps each index to the action of u in that position, $\lambda_u(i) = e_i$

In other words, a word u is a linearisation of an LPO G if there is a sequence of locations $l_0 \ldots l_n$ such that: i) the locations' labels match u ($\lambda(l_0) \ldots \lambda(l_n) = u$) and ii) the partial order depicted by G is not violated by the sequence of actions u. A linearisation of the LPO G provided in the previous paragraph is *pwd verify wait verifying ok reqCash getBalance*() *cash*.

Now we are ready to define the language of an LPO.

Definition 10 (Language of an LPO)**.** Given an LPO $G = \langle L, \leq, \lambda, \Sigma \rangle$, its language is defined as:

$$L_G = \{u \in \Sigma^* \mid u \text{ is a linearisation of } G\}$$

We define $|L_G|$ as the number of words in L_G, i.e. the number of linearisations of G.

For the LPO G discussed previously, as locations m_3 and m_4 that are mapped by λ to *verifying* and *wait* are not ordered, the LPO has two linearisations: { *pwd verify wait verifying ok reqCash getBalance*() *cash*, *pwd verify verifying wait ok reqCash getBalance*() *cash* }.

Sequence charts allow for the definition of co-regions [6]. A co-region is syntactically represented by a dashed line on the left of a group of messages. Co-regions delimit a scope in which the ordering of messages on a lifeline is not constrained. Examples of charts with co-regions can be seen in [24] or later on in Figures 4 and 11. Note that the presented sequence charts describe basic interactions and, unlike MSC and UML Interaction Diagrams, do not include constructs such as loops or alternatives.

The relation between the graphical syntax of a sequence chart and its corresponding abstrac syntax (LPO) is as follows (for more details refer to [6]): A message is an arrow pointing to a target instance. If the target is the same as the origin we say that the message is local. Messages occur at *points*. A location in a LPO is a set of points $\{p\}$ if the message is a self message in p or $\{p, q\}$ if there is an arrow from p to q. The location's label is the name of the message. Finally, two locations are directly ordered ($l_1 < l_2$) iff some point $p_1 \in l_1$ and some point $p_2 \in l_2$ are on the same lifeline and:

- p_1 is drawn above p_2, and
- p_1 and p_2 are not in the same co-region.

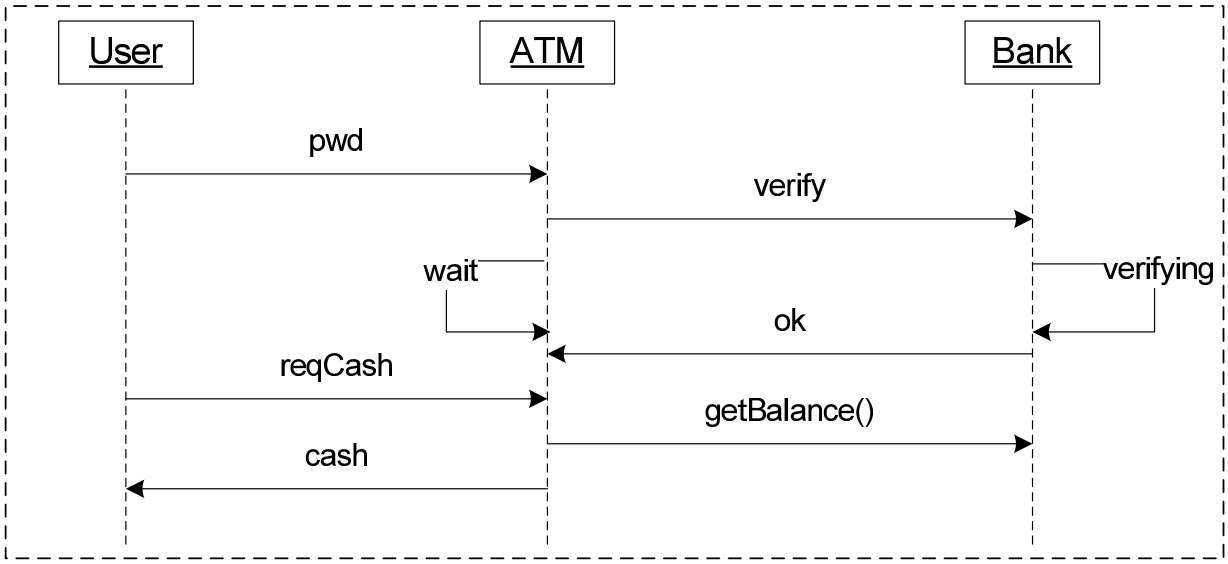


Figure 3: An existential live sequence chart (eLSC)

The relation $\leq$ is just the reflexive and transitive closure of $<$.

The MSC in Figure 2 has G, the previously discussed LPO, as its LPO and hence its language is defined as the linearisations of G. From now on, for the sake of simplicity, given a graphical syntax M of a scenario, and unless it is not obvious from the context, we may refer to it's abstract syntax as M

2.3 Live Sequence Charts - eLSC and uLSC

Many authors (e.g. [3, 4, 1, 2, 5]) have noted the limitations of the core scenario notation described above. One key issue is the limited expressiveness of a single sequence chart. Extensions have been developed to support sequence chart composition and provide control flow operations such as parallel, loops, concatenation, and alternatives. 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. [3] point out that the causal relation between events (messages and conditions) remains implicit in message sequence charts and that it can be beneficial to distinguish events that trigger a scenario from the events that occur in response to the trigger. In addition, they criticise the lack of distinction between universal and existential behaviour. Accordingly, they define a scenario-based description language based on sequence charts called Live Sequence Charts [8]. The core of LSCs, Constant LSCs [3], consist of two types of charts: (non-triggered) existential live sequence charts (eLSCs) and (triggered) universal live sequence charts (uLSCs).

An eLSCs is a sequence chart depicted in a dotted frame such as the one in Figure 3. We shall abstractly represent eLSCs as $\Diamond^{LSC}(B, \Sigma)$ where B is a sequence chart and $\Sigma \subseteq Act$ is the alphabet of the eLSC. The alphabet of the the eLSC is a superset of the message labels appering in B. The intuitive semantics of an eLSC is that there exists a trace of the system-to-be such that

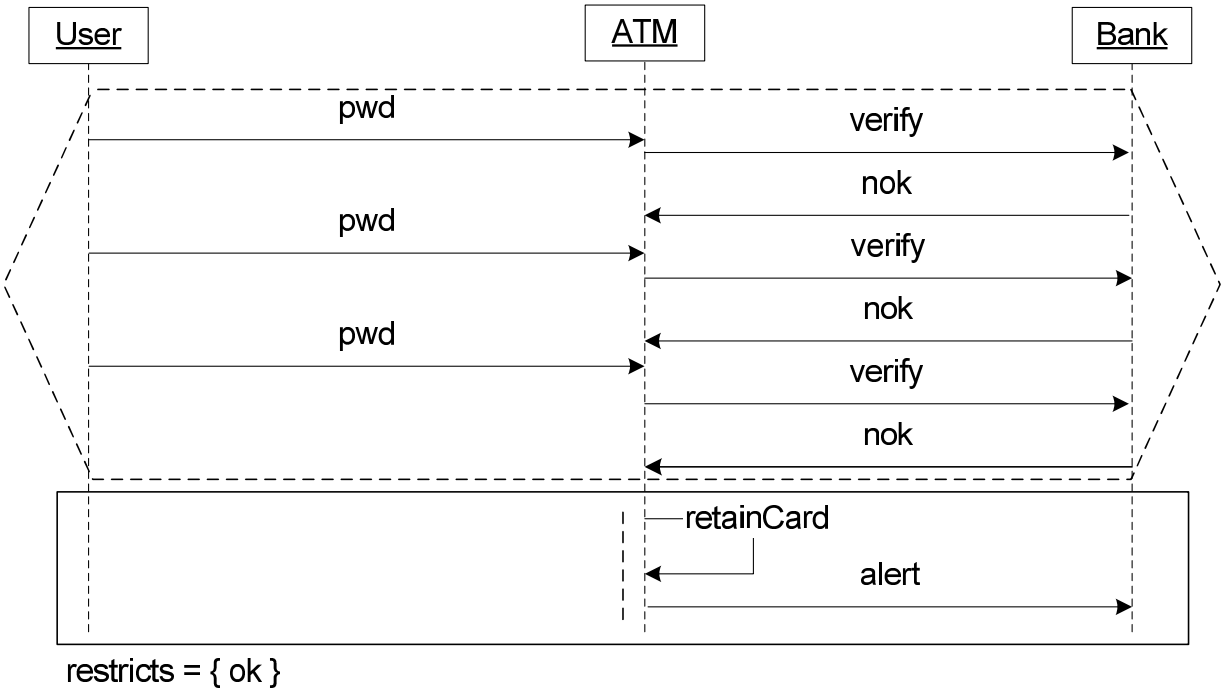


Figure 4: A universal live sequence chart (uLSC)

a portion of that trace, once projected onto Σ (Definition 11), is in L_B.

Definition 11 (Projection)**.** Let $w \in Act_\tau^*$, $\Sigma \subseteq Act_\tau$ and $t \in Act_\tau$. We define $\epsilon|_\Sigma$ as ϵ, and $tw|_\Sigma$ if $t \in \Sigma$ then tw' else $w'$where $w' = w|_\Sigma$

The purpose of including additional labels in the alphabet of an LSC is to restrict the occurrence of particular messages. For instance, the following sequence *pwd verify wait verifying ok reqCash getBalance*() *beep cash* ... is part of the language of the eLSC in Figure 3 with an alphabet that does not include *beep*, but would not be part of the language of the eLSC if *beep* were added to its alphabet. Syntactically, in any type of LSC, the actions that are part of the alphabet but do not appear in the charts (i.e. there are no messages with those labels) are included in a set *restricts* at the bottom of the charts as shown in Figure 4.

uLSCs consist of two sequence charts, a prechart and a main chart where the former is depicted above the latter (see Figure 4). We represent abstractly uLSCs as $\square^{LSC}(P, M, \Sigma)$ where P and M are sequence charts: the prechart and main chart respectively. $\Sigma \subseteq Act$ is the union of the message labels appearing in P and M and the restricts set. The intuitive semantics of a uLSC is that in every trace of the system-to-be, once projected onto the alphabet Σ it holds that for every occurrence of the prechart the main chart must immediately follow. Note that the main chart of a uLSC is depicted in a continuous frame to denote its universal nature in contrast to the dotted frame of eLSC (see Figure 3 and 4).

Consider the uLSC depicted in Figure 4, the language of its prechart contains one word: *pwd verify nok pwd verify nok pwd verify nok* and the language of the main chart contains two words (because of the co-region): *retainCard alert* and *alert retainCard*. The alphabet of the uLSC

is extended by the *restricts* clause and has the following actions {*pwd*, *verify*, *nok*, *retainCard*, *alert*, *ok*}. 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 and an alert must be sent to the bank. An example of a word 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.

Definition 12. *(Semantics of eLSC and uLSC)* Given an infinite word $w \in Act^\omega$ we say that,

- w satisfies an eLSC $E = \Diamond^{LSC}(B, \Sigma)$, written $w \models E$, if there is a decomposition uvw' of w such that $v|_\Sigma \in L_B$.
- w satisfies an uLSC $U = \Box^{LSC}(P, M, \Sigma)$, 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 words given by the words that satisfy the LSC: $L_S = \{w \in Act^\omega \mid w \models S\}$. In addition, given an LTS I with a set of traces L_I then,

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

In other words, an LTS satisfies an eLSC if at least one of its runs satisfies the existential scenario. Alternatively an LTS satisfies a uLSC if all its runs satisfy the universal scenario.

2.4 Fluents

The triggered scenario-specification language which we introduce in Section 3 has conditions. These conditions are in the form of Fluent Propositional Logic which supports natural specification in event-based descriptions such as scenarios.

A fluent [25] Fl is defined by a pair of sets and a boolean value: $Fl = \langle I_{Fl}, T_{Fl}, Init_{Fl}\rangle$. I_{Fl} is the set of initiating actions and T_{Fl} the set of terminating actions such that $I_{Fl}, T_{Fl} \subseteq Act$ and $I_{Fl} \cap T_{Fl} = \emptyset$. A fluent may be initially *true* ($\top$) or *false* ($\bot$) as indicated by $Init_{Fl}$. Every action $a \in Act$ induces a fluent, namely, $a = \langle\{a\}, Act \setminus \{a\}, \bot\rangle$. Finally the alphabet of a fluent is the union of its terminating and initiating actions.

Let $\pi = a_1 a_2 \ldots a_i \in Act^*$, π satisfies a fluent Fl, denoted $\pi \models Fl$, if and only if one of the following conditions holds:

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

In other words, a fluent holds after a word if and only if it holds initially or some initiating action has occurred and, in both cases, no terminating action has yet occurred.

$$\begin{array}{lcl} \pi \models \phi \vee \psi & \triangleq & (\pi \models \phi) \vee (\pi \models \psi) \\ \pi \models \neg\phi & \triangleq & \neg(\pi \models \phi) \end{array}$$

Figure 5: Semantics of satisfaction operator.

Let $\mathscr{F}$ be the set of all possible fluents defined over Act. $Fl \in \mathscr{F}$ is a Fluent Propositional Logic (FPL) formula and other FPL formulas are defined inductively using the standard boolean connectives as shown in Figure 5.

We will use the logic $(\Upsilon, \mathscr{I}, \models)$ where Υ are the formulas in FPL, $\mathscr{I}$ is an interpretation for the fluents appearing in those formulas, and $\models \subseteq \mathscr{I} \times \Upsilon$ a model relation where $i \models \phi$ means ϕ is true under interpretation i. The interpretation is just the valuation of the fluents $i : \mathscr{F} \to \{\top, \bot\}$ and more complex FPL formulas are interpreted as depicted in Figure 5.

The valuation of the fluents after a word is known through a function defined over a set of fluents FS that relates sequences of actions with states.

Definition 13 (State function defined over FS)**.** A state function defined over the set of fluents FS is a function $\zeta : Act^* \to (FS \to \{\top, \bot\})$.

From a set of fluents FS, the state function ζ derived from FS is defined recursively using the initial values of the fluents:

Definition 14 (State function derived from FS)**.**

$$\begin{array}{ll} \zeta(\epsilon) & = \{x \mapsto Init_x \mid x \in FS\} \\ \zeta(wa) & = \{x \mapsto update(x, \zeta(w)(x), a) \mid x \in FS\} \end{array}$$

where $update(x, b, a)$ is $\top$ if $a \in I_x$, $\bot$ if $a \in T_x$, and b otherwise

If $FS = \emptyset$ then we note the state function derived from FS as $\zeta^{\emptyset}$ and, for any word w, $\zeta^{\emptyset}(w)$ is the empty function. The valuation after a sequence of actions w is noted ζ_w such that for any $z \in Act^*$ $\zeta_w(z) = \zeta(wz)$. Note that if ζ is the state function derived from FS then, for instance, ζ_w is a state function over FS. Finally, we will omit mentioning the set of fluents when it is clear from the context.

3 Triggered Scenarios

In this section we propose a triggered scenario specification language that is capable of describing both conditional existential and conditional universal scenarios. Informally, a conditional scenario is an assertion that has the following structure: if p occurs then m occurs, where p and m describe system behaviour. An existential interpretation of a conditional scenario requires that if p occurs, then m *may* occur while a universal interpretation will require that if p occurs then m *must* occur.

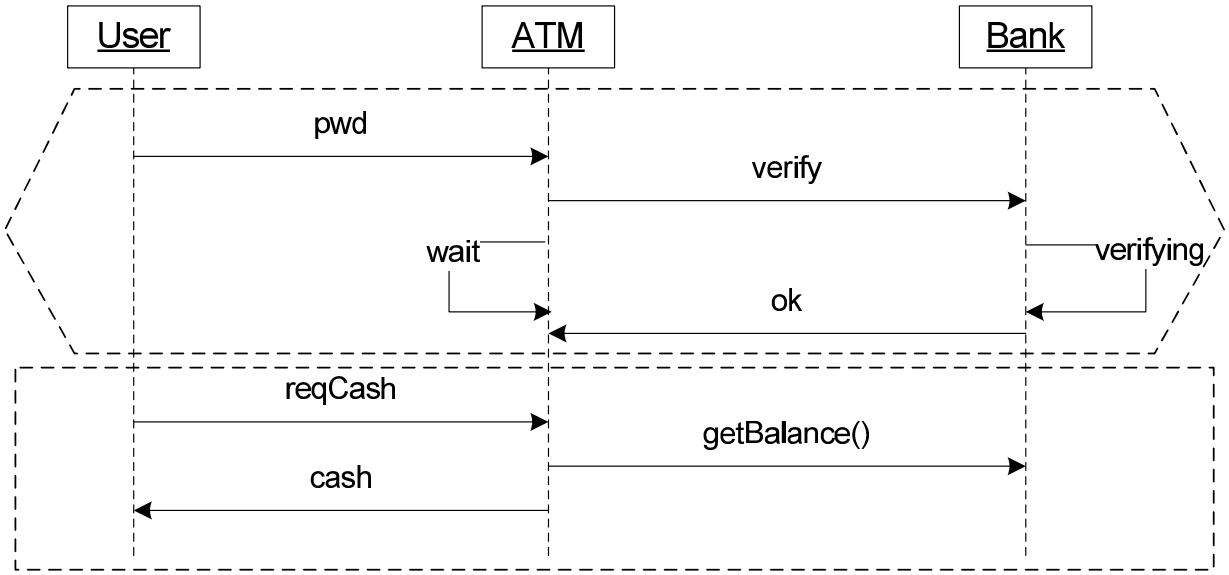


Figure 6: An eTS

Conditional universal scenarios are commonly used. An example is a statement such as "if, after inserting the card into the ATM, the user inputs an invalid password three times in a row then the ATM *must* retain the card and alert the bank". Such a statement can be described with the uLSC of Figure 4.

Conditional existential scenarios are also commonly used, notably in use case style specifications. For instance, "if the user inserts a valid card into the ATM, and then enters the correct password, she/he *may* request cash and have it dispensed by the ATM". The existential interpretation does not prohibit behaviour such as requesting a balance, while a universal interpretation would. This conditional existential statement can be formalised with an Existential Triggered Scenario (eTS) as depicted in Figure 6.

We now define a language of triggered scenarios that supports existential and universal interpretations; a detailed comparison between these triggered scenarios and LSCs is given in Section 6. Triggered Scenarios (TS) consist of two sequence charts (as defined in Section 2.2): a trigger and a main chart. The former is drawn inside a dashed diamond above the latter. The trigger may have conditions in the form of FPL formulas. The scenario alphabet is the union of actions appearing as message labels in the trigger and the main chart, in fluent definitions and in the *restricts* that may appear at the bottom of a scenario as shown in Figure 4.

The intuitive semantics of eTS is that every time that the trigger holds, the system-to-be *must be able* to exhibit *all* the behaviour in the main chart. In case of the eTS in Figure 6, every time the user logs in he/she *must be able* to withdraw money. The semantics of eTS cannot be formally defined in terms of words; it must instead be done using *computation trees* (recall Definition 7). Informally, a tree satisfies an eTS if for every branch in which the triggers occurs, this is immediately followed by a branch for every behaviour described in the main chart.

Consider Figure 7 where a portion of an infinite tree satisfying the eTS in Figure 6 is depicted.

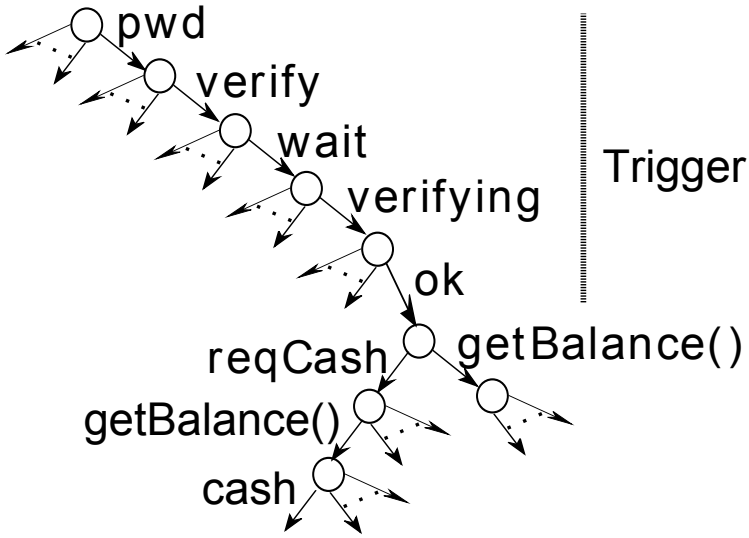


Figure 7: Part of an infinite computation tree satisfying eTS in Figure 6

The trigger has occured at the state reached by the transition labelled *ok*. From this state, in accordance with the eTS, there is a branch defining a word that satisfies the main chart. The fact that from the same state there is a branch that does not satisfy the main chart is irrelevant for satisfying an existential triggered scenario.

The intuitive semantics of universal triggered scenario (uTS) is that every time that the trigger holds, the system-to-be *must be able* to exhibit *all* the behaviour in the main chart and *only* that behaviour. The semantics of uTS must also be defined over computation trees. Informally a computation tree satisfies an uTS if for every branch where the trigger holds, not only is immediately followed by a branch for every behaviour described in the main chart, but also all branches exhibit behaviour described in the main chart.

Let us consider if the partially depicted tree of Figure 7 satisfies the triggered scenarios of Figure 6 under a universal interpretation: As before, the trigger holds at the state reached by the transition labelled *ok*. From this state, there is a branch defining a word that satisfies the main chart. However, from the same state a branch that does not satisfy the main chart. Consequently, the tree does not satisfy the uTS.

Note how if there are several linearisations of the main chart then *all* should (in the case of eTS) or must (in the case of uTS) be present. This is in line with semantics like the one given for MSC in [26] and High Level MSC in [27] where every linearisation of the charts should be present in an implementation.

The examples of triggered scenarios given so far to exemplify their semantics do not include conditions. Semantics of conditions requires some additional explanation. As stated before: triggers may include conditions in the form of FPL formulas. They are drawn in rounded boxes and can cover one or more instances. For the trigger to hold, not only must a sequence of messages that corresponds to an ordering of the trigger occur but also conditions must be satisfied as soon as they are reached. For example the trigger in Figure 8 is satisfied when a message *a*

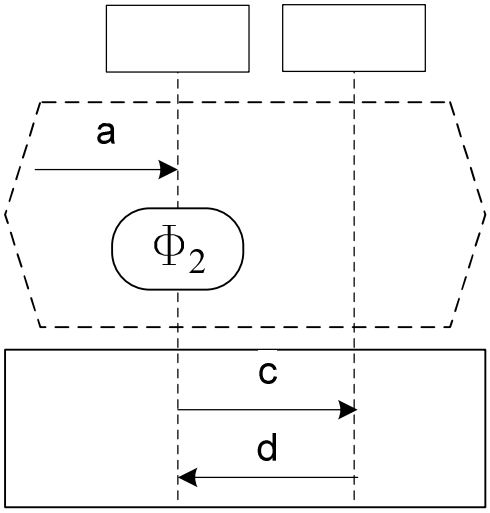


Figure 8: A trigger with one message and one condition

(short for a message labelled a) occurs and, immediately after that instant, Φ_2 is true.

In the remainder of this section we formally define Triggered Scenarios giving their abstract syntax and semantics.

3.1 Syntax

The main chart is abstractly represented by an LPO. Triggers are abstractly represented by Labelled Partial Order with Conditions (LPOC), an extension to LPOs that includes conditions and formalises the intuitions given paragraph above.

Definition 15 (LPO with Conditions (LPOC))**.** An LPOC is a tuple $\langle L, \leq, \lambda, \Sigma, \Psi \rangle$ where

- L is a finite set of locations
- $\leq \subseteq L \times L$ is a partial order relation over L.
- $\lambda : L \to \Sigma \cup \Psi$ is a labeling function.
- Ψ is a set of FPL formulas over the alphabet Σ.

As with LPO, given an LPOC T we define $|T|$ as the number of locations in T.

The relation between the diagrammatic representation of triggers and LPOCs is a simple extention to that of sequence charts and LPOs (see Section 2.2 and [6]). Each condition is associated with a location that has one point per lifeline that the condition covers in the trigger. Diagrammatically a condition defines a segment. As with messages, condition segments do not cross each other nor with messages. For example the condition Φ_2 in the trigger of Figure 10 forms a segment that covers two instances and so its associated location will contain two points.

Thus, the condition Φ_2 will precede (resp. follow) any message or condition that covers either lifeline and appears below (resp. above) its segment. Φ_2 precedes message c and follows message b but is not ordered with respect to message a or condition Φ_1. For example the trigger of the scenario in Figure 10 defines an LPOC $T = \langle L, \leq, \lambda, \Sigma \rangle$ where $L = \{m_1, m_2, m_3, c_1, c_2\}$, $\leq$ is the reflexive and transitive closure of $\{(m_1, c_1), (c_1, m_3), (c_2, m_3), (m_2, c_2)\}$, and $\lambda = \{(m_1, a), (m_2, b), (m_3, c), (c_1, \phi_1), (c_2, \phi_2)\}$.

We now define linearisations of LPOC similarly to that of LPO except that now linearisations must guarantee that conditions must appear as early as the partial order permits.

To define the linearisations of an LPOC $T = \langle L, \leq, \lambda, \Sigma, \Psi \rangle$ we are going to use the linearisation of its associated LPO $T_{LPO} = \langle L, \leq, \lambda, \Sigma \cup \Psi \rangle$. The FPL formulas and messages of T are treated equally in T_{LPO}. A linearisation of T_{LPO} is then a combination of messages and formulas conforming to the partial order in T. The intuitive idea is that $w \in \Sigma^*$ with a state function ζ defined over the set of fluents present in T is a linearisation of T iff there exists a $v = a_0 \dots a_n \in (\Sigma \cup \Psi)^*$ such that v is a linearisation of T_{LPO} and (i) $w = v|_\Sigma$ (ii) if a_j is a formula then the projection onto Σ of v up to a_j satisfies a_j (iii) v is a linearisation of T_{LPO} such that the conditions appear as soon as possible with respect to messages.

Definition 16 (linearisation of an LPOC)**.** Let T be an LPOC $\langle L, \leq, \lambda, \Sigma, \Psi \rangle$, ζ a state function defined over the set of fluents present in T, and $w \in \Sigma^*$. A tuple $\langle w, \zeta \rangle$ is a linearisation of T if and only if there exists a linearisation $v = \lambda(l_0) \dots \lambda(l_n)$ of LPO $T_{LPO} = \langle L, \leq, \lambda, \Sigma \cup \Psi \rangle$ such that

i) Messages and formulas are ordered according to T:
$w = v|_\Sigma$

ii) The conditions are satisfied:
$\forall j \in \{0, \dots, n\} \cdot \lambda(l_j) \in \Psi \implies \zeta(\lambda(l_0) \dots \lambda(l_j)|_\Sigma) \models \lambda(l_j)$

iii) Conditions appear as soon as possible:
$\forall j \in \{0, \dots, n\} \cdot \lambda(l_j) \in \Psi$:
if $(\nexists i \in \{0, \dots, n\} \cdot \lambda(l_i) \in \Sigma \wedge l_i \leq l_j)$ then $\forall k \in \{0, \dots, n\} \cdot \lambda(l_k) \in \Sigma \implies j < k$
else $\nexists k \in \{0, \dots, n\} \cdot \lambda(l_k) \in \Sigma \wedge posConditionEnabled < k < j$
where $posConditionEnabled = \max_i\{i \in \{0, \dots, n\} \mid l_i \leq l_j \wedge \lambda(l_i) \in \Sigma\}$

To exemplify the above definition, consider the trigger in Figure 9. The linearisations of the LPO associated to the LPOC of the trigger are { $a\Phi_1 b\Phi_2 c$, $a\Phi_1 cb\Phi_2$, $b\Phi_2 a\Phi_1 c$, $ba\Phi_2\Phi_1 c$, $ab\Phi_2\Phi_1 c$, $ba\Phi_1\Phi_2 c$, $ab\Phi_1\Phi_2 c$ }. However, the last four linearisations do not satisfy condition (iii): The location of Φ_2 is preceded by the location of b in the partial order relation over L, hence (iii) forbids a message between b and condition Φ_2. In other words, as soon as b occurs Φ_2 should be tested before any other message occurs. This removes $ba\Phi_2\Phi_1 c$ and $ba\Phi_1\Phi_2 c$. Analogously, according to the partial order relation over L, the location of Φ_1 is preceded by the location of a then (iii) removes linearisations $ab\Phi_2\Phi_1 c$ and $ab\Phi_1\Phi_2 c$ where b is in between a and Φ_1. Therefore linearisations of the LPOC when projected onto Σ (condition (i)) must result in one of the first three LPO linearisations: $\{a\Phi_1 b\Phi_2 c, a\Phi_1 cb\Phi_2, b\Phi_2 a\Phi_1 c$ }.

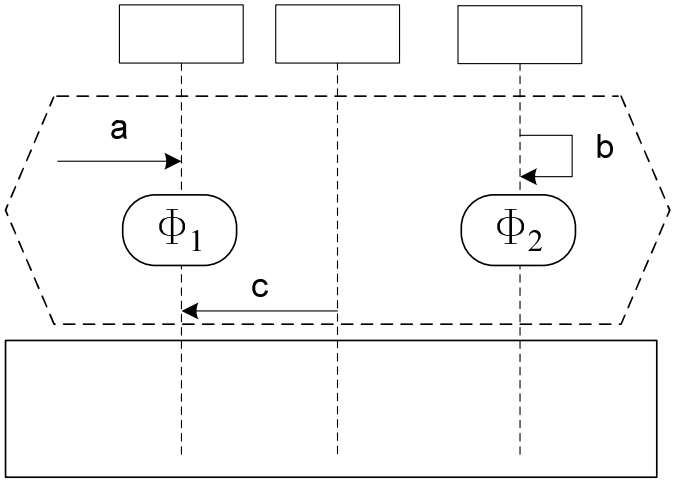


Figure 9: A trigger where Φ_2 only affects one instance

To exemplify condition (ii), which is related to the satisfaction of conditions, we must define the conditions and fluents. Let ϕ_1 and ϕ_2 be fluents defined as $\langle\{c\},\{b\},\top\rangle$ and $\langle\{c\},\{a\},\top\rangle$ respectively, and Φ_1 and Φ_2 be conditions defined by formulas ϕ_1 and ϕ_2 respectively. Note that Definition 16 only requires the state function to be defined over the fluents in T. In this case there are two fluents which allow for the definition of four different state functions over the fluents in T. If ζ^T is the state function derived from the set of fluents present in T then $\zeta^T(\epsilon)(\phi_1) = \bot$ and $\zeta^T(\epsilon)(\phi_2) = \bot$. The remaining three state functions can be described, for example, as ζ^T_a, ζ^T_b and ζ^T_{ab}. $\zeta^T_a(\epsilon)(\phi_1) = \top$ and $\zeta^T_a(\epsilon)(\phi_2) = \bot$, $\zeta^T_b(\epsilon)(\phi_1) = \bot$ and $\zeta^T_b(\epsilon)(\phi_2) = \top$, and $\zeta^T_{ab}(\epsilon)(\phi_1) = \bot$ and $\zeta^T_{ab}(\epsilon)(\phi_2) = \bot$. The intuitive idea on why it is needed to consider other state functions besides the initial one derived from the set of fluents is to capture the fact that the valuation of the fluent changes with the evolution of the system.

To continue with the example let's consider ζ^T. For these conditions and state function, neither the first nor third LPO linearisations satisfy item (ii) of the above definition. Consider $a\Phi_1 b\Phi_2 c$, it does not hold that $\zeta^T(ab) \models \Phi_2$ as a makes ϕ_2 false. Equally, LPO linearisation $b\Phi_2 a\Phi_1 c$ does not satisfy (ii) as $\zeta^T(ba) \not\models \Phi_1$.

Consequently, for ζ^T, the only LPO linearisation that satisfies items (ii) and (iii) is $a\Phi_1 cb\Phi_2$ as $\zeta^T(a) \models \Phi_1$ and $\zeta^T(acb) \models \Phi_2$, consequently (from item (i)) $\langle acb, \zeta^T\rangle$ is a linearisation of the LPOC for Figure 9. To find the remaining linearisations a similar procedure has to be performed with each one of the remaining state functions.

Recall that changing the lifelines covered by a condition modifies the LPOC and consequently its linearisations. For instance consider the trigger in Figure 10 that differs from the trigger in Figure 9 only in condition Φ_2 that now covers two lifelines. This modifies the partial order so that c must come after Φ_2 therefore reducing the linearisations of the associated LPO that satisfy condition (i) to $\{a\Phi_1 b\Phi_2 c, b\Phi_2 a\Phi_1 c\}$.

The language of an LPOC is defined as the set of all pairs $\langle w, \zeta\rangle$ that are its linearisations:

Definition 17 (Language of an LPOC)**.** Let T be an LPOC $T = \langle L, \leq, \lambda, \Sigma, \Psi\rangle$ its language is

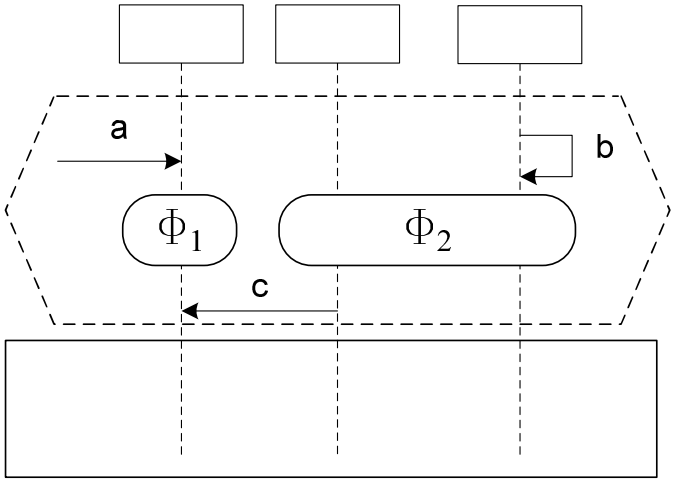


Figure 10: A trigger where Φ_2 affects two instances

defined as:

$$L_T = \{\langle \alpha, \zeta \rangle \mid \langle \alpha, \zeta \rangle \text{ is a linearisation of } T\}$$

As the linearisations of T contain only state functions defined over the fluents present in T, then L_T is finite. As with LPO, given an LPOC T we define $|L_T|$ as the number of elements (pairs of words and state functions) in L_T.

Finally, we define a satisfaction relation between a word with a state function and triggers. Intuitively, a word and the state function derived from the set of fluents affecting the trigger satisfies the trigger if a suffix of the word together with the state function is part of the trigger's language.

Definition 18. Given the state function ζ^T derived from the fluents present in an LPOC T and a word α we say that α with ζ^T satisfies T (written $\alpha, \zeta^T \models T$) if and only if $\exists uv \cdot \alpha = uv \ \wedge \langle v|_\Sigma, \zeta^T_u \rangle \in L_T$.

Having defined LPOCs and their linearisations we now proceed to formally define eTS and uTS as tuples of an LPOC (trigger), an LPO (main chart) and an alphabet. In the next section we provide a semantics for both triggered scenarios.

Definition 19. An Existential Triggered Scenario (eTS) is a tuple $E = \Diamond(T, M, \Sigma)$ where T (the trigger) is an LPOC with alphabet Σ and M (the main chart) is an LPO with alphabet Σ.

Definition 20. A Universal Triggered Scenario (uTS) is a tuple $U = \Box(T, M, \Sigma)$ where T (the trigger) is an LPOC with alphabet Σ and M (the main chart) is an LPO with alphabet Σ.

In both cases Σ, the scenario's alphabet, is the union of: the actions appearing as message labels in T and M, the alphabet of the fluents (that is their initiating and terminating actions) appearing in T, and the actions in the *restricts* set.

3.2 Semantics

As explained informally at the beginning of the Section, the semantics of TS is given in terms of computation trees. If a branch of the tree that starts at the initial state and ends at state n defines a word that together with the state function derived from the fluents satisfies the trigger, then (both in the case of eTS and uTS) for each word $m \in L_M$ at least one branch starting at n must define a word that when projected on Σ is equal to m. In the case of uTS there is another condition: every branch starting at n defines a word that when projected on Σ is in L_M. Formally:

Definition 21. A computation tree satisfies the eTS $E = \Diamond(T, M, \Sigma)$ if and only if for every branch b starting in the tree's initial state the following holds where ζ^T is the state function derived from the fluents present in T, s is the end state of b and w is the word defined by b:
$w, \zeta^T \models T \implies \forall m \in L_M \cdot \exists b'$ branch starting at s defining a word w' such that $w'|_\Sigma = m$.

Definition 22. A computation tree satisfies the uTS $U = \Box(T, M, \Sigma)$ if and only if for every branch b starting in the tree's initial state the following holds where ζ^T is the state function derived from the fluents present in T, s is the end state of b and w is the word defined by b:

- $w, \zeta^T \models T \implies \forall m \in L_M \cdot \exists b'$ branch starting at s defining a word w' such that $w'|_\Sigma = m$.
- $w, \zeta^T \models T \implies \forall b'$ complete branch starting at s defining a word w' then $\exists uv \cdot uv = w' \wedge u|_\Sigma \in L_M$.

Finally, we define the satisfaction relation between LTS and TS as the satisfaction of the LTS's computation tree of the TS.

Definition 23. An LTS I satisfies a Triggered Scenario Sc (written $I \models Sc$) iff the computation tree of I satisfies Sc.

One point worth mentioning is that of vacuous [28] triggered scenarios. A vacuous triggered scenario is one which is only satisfied by computation trees in which the trigger never occurs. There are two causes for this. First, it is possible to define a trigger which is not satisfiable by any computation tree. An example of this would, for instance, any trigger that has an unsatisfiable condition. Another more subtle situation is a condition that, due to the messages and conditions that precede it in a trigger, cannot be satisfied. An example of the latter would be Figure 9 with $\Phi_1 = \phi_1$ and fluent ϕ_1 defined as $\langle\{c\}, \{a\}, \bot\rangle$. In this case, every time a occurs, Φ_1 will be false. A second cause for vacuity is, for uTS, when the main chart specifies behaviour that is inconsistent with the trigger. Informally, this may be the case if a uTS triggers itself: the main chart requires a certain behaviour uv where u satisfies its trigger but where v does not (or cannot be extended to) satisfy the main chart. Checking for vacuity of TS is a special case of the much studied more general problem and can be done following [28]. In the remainder of this paper we assume all TS to be non-vacuous.

4 MTS Synthesis

In this section we define synthesis algorithms that construct behaviour models in the form of Modal Transition Systems (MTS) from non-vacuous TSs.

In general, the scenario synthesis problem consists of 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 modelling formalism chosen as a target of the synthesis, and the various additional constraints that can be imposed such as in distributed synthesis (e.g. [2]).

A stronger requirement for the synthesis is that the resulting model characterises, 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 (see [7, 17, 5]).

It is convenient to characterise all behaviour models that satisfy a given scenario-based description in one operational model 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.

We now present an algorithm that given a non-vacuous TS Sc with alphabet Σ produces an MTS model M that characterises all LTS that satisfy the scenario; formally $I@\Sigma \in \mathcal{I}[M] \Leftrightarrow I \models Sc$. This entails that MTS refinement preserves the semantics of TSs and that MTS merge provides a composition mechanism for TS. In other words, the synthesis of an MTS from a set of TS can be defined as merging the MTS synthesised from each TS.

There are two key issues to take into account when synthesising an MTS from a Triggered Scenario. The first is that the MTS must observe but not restrict behaviour and detect when a sequence of actions that satisfies the trigger has occurred. The second, is that once the trigger has been satisfied, the MTS must ensure certain behaviour from that point on. If the synthesis is from a uTS then the MTS must guarantee all traces in the main chart's language and also that only traces in the language of the main chart can occur. In the case of eTS, the MTS must guarantee all traces in the main chart's language but allow all other behaviour.

The differences in the semantics of eTS and uTS makes the synthesis algorithms for each sufficiently different to necessitate presenting them separately. We start with eTS and then go on to uTS.

4.1 Synthesis from eTS

We first run through an example to illustrate how an MTS characterises all implementations that satisfy an eTS and then we present the synthesis algorithm.

4.1.1 Running example

Consider the eTS in Figure 11 with trigger T and main chart M. Given that there are no conditions in T there is then only one possible state function, the empty function which we note $\zeta^{\emptyset}$. Hence

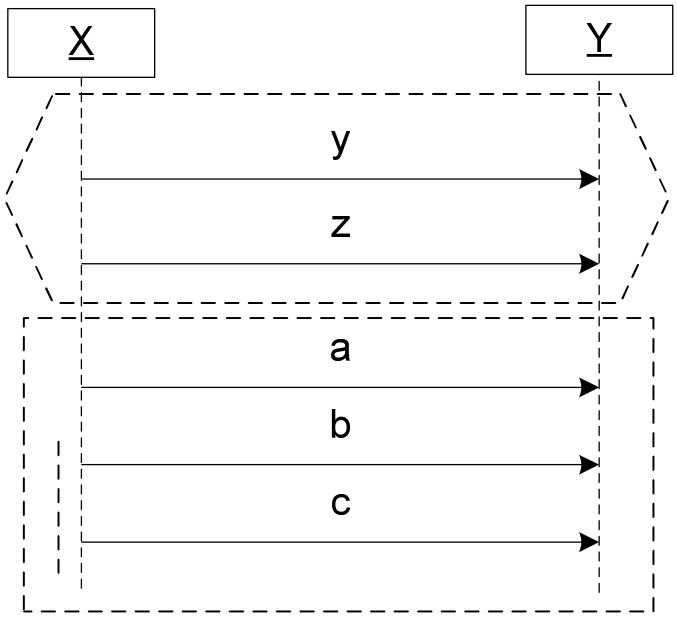


Figure 11: The eTS used as the synthesis algorithm's running example

we consider $L_T = \{\langle yz, \zeta^{\emptyset}\rangle\}$. The main chart's language is $L_M = \{abc, acb\}$ and the alphabet is $\Sigma = \{a, b, c, y, z\}$.

The algorithm that we introduce in the next section produces the MTS in Figure 12 (unreachable states are not shown) for the eTS discussed in the previous paragraph. All implementations of the MTS satisfy the eTS and all LTS that satisfy the eTS are implementations of the MTS. Note that in Figure 12 states are annotated with the data structure (a tuple) that the algorithm uses to represent states. An explanation of the state's structure will be given in Section 4.1.2. States that are not reachable from the initial state are not shown.

The MTS in Figure 12 guarantees that any of its traces that end with the sequence of actions yz lead to state 2. In other words, when the trigger of the eTS is satisfied, the MTS will be in state 2. Furthermore, note that any trace that does not ever satisfy the trigger will only cover maybe transitions leading to states 0 and 1. That is, the MTS does not require implementations to provide any specific behaviour if the trigger of the eTS is not satisfied.

From state 2, reached if and only if the trigger holds, there are two paths of required transitions. Each path represents a word in L_M. Intuitively the state where the trigger holds has some *obligations*: the words in the main chart's language. In order to make all refinements of the synthesised model satisfy the eTS we need a required path for each obligation. Thus, the required transitions from $(2, a, 3)$, $(3, c, 4)$, $(4, b, 0)$, $(2, a, 5)$, $(5, b, 6)$ and $(6, c, 0)$.

Although states 2 through 6 have outgoing required transitions to guarantee that all implementations of the MTS will provide the behaviour of the eTS's main chart when the eTS's trigger has occurred, these states also have maybe transitions. These transitions ensure that any LTS that provides other behaviour in addition to that of the main chart once the trigger is satisfied is also

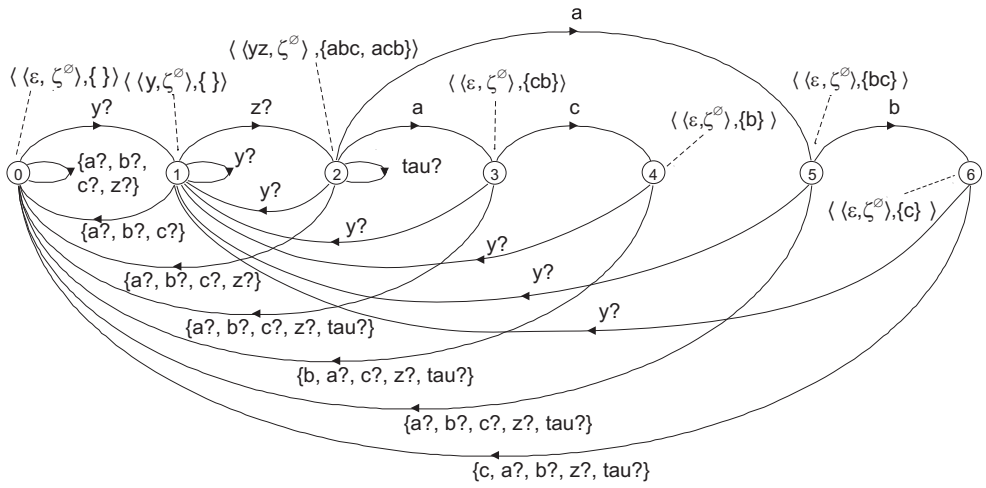


Figure 12: MTS synthesised from the eTS running example with states annotated with the state's structure (unreachable states not shown)

an implementation of the MTS.

For example, the outgoing transitions from state 3, labelled y, a, and b, are needed in Figure 12 to allow for the implementation LTS in Figure 13 which satisfies the eTS in Figure 11. Without the MTS's maybe transitions along the required paths, state 3 of the LTS, in which a, b and τy are possible, does not have a counterpart in the MTS. Furthermore, state 3 of the LTS has a τ transition to 0 where c and b are no longer possible. Implementations such as this last one, that abort the completion of the main chart through a τ transition, are captured using the maybe τ transitions along the required paths of the MTS. For instance the MTS has a τ transition from 3 and 5 to 0 where there is no required behaviour.

Note that the MTS in Figure 12 has a non-deterministic choice on state 2 for action a. This is needed to capture all implementations that satisfy the scenario. For example if we join states 3 and 5, making the choice on a deterministic, the LTS in Figure 14 would not be an implementation of the MTS; however the LTS satisfies the eTS. The reason that the LTS is not an implementation of the deterministic MTS is that the a transition in the deterministic MTS leads to a state in which both b and c are required, however such a state does not exist in the LTS. In summary, the non-determinism on a in Figure 12 is needed to guarantee that it characterises all implementations that satisfy the eTS.

4.1.2 Synthesis

The synthesis strategy of the algorithm presented below is to represent each state of the MTS with a tuple that represents what portion of the trigger of an eTS has occurred and what obligations, in terms of required behaviour, the state has. In other words, each state of the synthesised MTS is represented by a structure that has two parts: the recognised trigger prefix and the state's pending obligations. The structure of the states will be formally defined later in Definition 29.

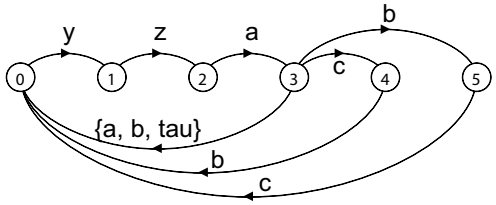


Figure 13: An LTS satisfying the scenario in Figure 11

Pending obligations are suffixes of words in the language of the main chart of a TS. Pending obligations for the Figure 11 are *b*, *c*, *bc*, *cb*, *abc* and *acb*.

A *recognised trigger prefix* is a pair $\langle \alpha, \zeta \rangle$ such that $\langle \alpha, \zeta \rangle \in \mathit{prefixes}(L_T)$ where T is the trigger of a TS and *prefixes*() is defined as follows:

Definition 24 (Prefixes)**.** Let T be the trigger of a Triggered Scenario. We define $\mathit{prefixes}(L_T)$ as $\{\langle \alpha', \zeta \rangle \mid \exists \langle \alpha, \zeta \rangle \in L_T \ \wedge \exists \alpha'' \cdot \alpha = \alpha' \alpha''\}$.

Recall that, as the possible state functions are defined over the fluents present in T, L_T is finite and therefore so is $\mathit{prefixes}(L_T)$.

Consider, for instance, state 2 of Figure 12 which has $\langle yz, \zeta^{\emptyset} \rangle$ as its recognised trigger prefix and words *abc* and *acb* as pending obligations. This means that any trace of the form $w_0 \ldots w_n yz$ in which the state function after w_n is $\zeta^{\emptyset}$ will lead to state 2 and that from state 2 there are exactly two sequences of required transitions that can be taken, one labelled *a*, *b*, *c* and the other labelled *a*, *c*, *b*.

State 3 has an empty recognised trigger $\langle \epsilon, \zeta^{\emptyset} \rangle$ which is consistent with the fact that it can only be reached through an *a* transition. In other words, when in state 3, the longest prefix of a word in L_T is the empty word as no prefix of L_T includes *a*. The only obligation of state 3 is *cb*, corresponding with the fact that the *a* of the obligation *acb* of state 2 will have occurred. Note that state 5 is similar to 3 but has an obligation *bc*.

More generally, the algorithm builds an MTS that guarantees (see Property 1) that for every trace of the MTS that leads to a state $s = \langle rtp, \Theta \rangle$, the longest suffix of the trace that corresponds to a prefix of the trigger is *rtp* (item 1 of Property 1). The algorithm also guarantees that from every state, the outgoing paths of required transitions correspond exactly to the state's obligations (implied by item 2 of Property 1) and that any word over the eTS alphabet is a possible trace from every state (implied by item 3 of Property 1).

Definition 25 (Significant suffix)**.** Let $T = \langle L, \leq, \lambda, \Sigma, \Psi \rangle$ be a LPOC, $\langle \gamma, \zeta \rangle$, $\gamma \in \Sigma^*$ and ζ a state function defined over the fluents present in T. We define $\mathit{sigSuf}(\langle \gamma, \zeta \rangle)$ to be the tuple with the longest first element (i.e. γ') in
$\{\langle \gamma', \zeta_\alpha \rangle \mid \gamma = \alpha\gamma' \wedge (\langle \gamma', \zeta_\alpha \rangle \in \mathit{prefixes}(L_T) \vee \gamma' = \epsilon)\}$

Note that in the above definition, if there is no suffix of $\langle \gamma', \zeta_\alpha \rangle$ that is a prefix of T then $\langle \epsilon, \zeta_\alpha \rangle$ is considered even in the case that $\langle \epsilon, \zeta_\alpha \rangle$ is not a prefix of L_T.

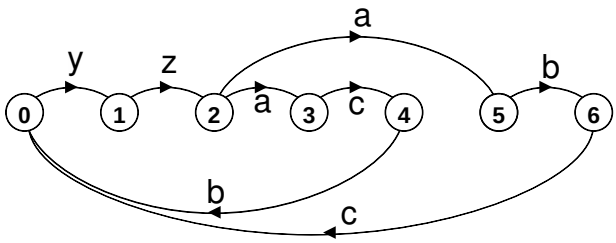


Figure 14: Another LTS satisfying the scenario in Figure 11

Property 1 (Invariant). Let W be the MTS synthesised from an eTS with trigger T, main chart M and alphabet Σ. For all trace π such that $W \xrightarrow{\pi}_{\mathrm{p}} W_s$ with $s = \langle rtp, \Theta \rangle$ then:

1. $sigSuf(\langle \pi|_\Sigma, \zeta^T \rangle) = rtp$, where ζ^T is the state function derived from the fluents present in T
2. $\forall \ell \in \Sigma \cdot W_s \xrightarrow{\ell}_{\mathrm{r}} \Leftrightarrow \exists \theta \cdot \ell\theta \in \Theta$
3. $\forall \ell \in \Sigma \cdot W_s \xrightarrow{\ell}_{\mathrm{p}}$

The MTS synthesis procedure adds a transition labelled ℓ between two states if and only if updating the recognised trigger prefix and obligations of the transition's source with ℓ is compatible with the recognised trigger prefix and obligations of the transition's target.

Updating the recognised trigger prefix $\langle \alpha, \zeta \rangle$ with an action ℓ returns the longest suffix of $\langle \alpha\ell, \zeta \rangle$ that is a prefix of the trigger. For instance updating $\langle y, \zeta^\emptyset \rangle$ with z yields $\langle yz, \zeta^\emptyset \rangle$, updating it with a yields $\langle \epsilon, \zeta^\emptyset \rangle$ and updating it with y yields $\langle y, \zeta^\emptyset \rangle$. Note that in the formalisation below, *signSuf* updates values of fluents in ζ with a prefix of α that may be dropped.

Definition 26 (Update Recognised Trigger Prefix)**.**

$$\begin{aligned} updateTrig(\langle \alpha, \zeta \rangle, \tau) &= \langle \alpha, \zeta \rangle \\ updateTrig(\langle \alpha, \zeta \rangle, \ell) &= sigSuf(\langle \alpha\ell, \zeta \rangle) \end{aligned}$$

Updating obligations based on a transition labelled ℓ is slightly more complicated. If the transition is required (the sets of required an possible transitions are formally defined in Definition 29), then the update is computed as the union of the *new obligations* that are contracted by taking the transition and a remaining or *inherited* obligations of the source state after taking the transition. New obligations are the words in L_M if the update of the recognised trigger prefix results in a $\langle \alpha, \zeta \rangle \in L_T$ and $\emptyset$ otherwise. An inherited obligation is the result of taking exactly one of the words in the source state's obligations that starts with ℓ and removing the initial ℓ.

Definition 27 (Update Obl. Upon a Req. Trans. eTS)**.** Let ℓ be a label in Σ. We define $updateOblURReqT(\langle rtp, \Theta \rangle, \ell)$ as the set

$$\begin{aligned} \{\ \Theta' \mid\ & \exists w \cdot \ell w \in \Theta \wedge \Theta' = inhObl \cup newObl \wedge (w = \epsilon \implies inhObl = \emptyset) \wedge \\ & (w \neq \epsilon \implies inhObl = \{w\}) \wedge (updateTrig(rtp, \ell) \in L_T \implies newObl = L_M) \wedge \\ & (updateTrig(rtp, \ell) \notin L_T \implies newObl = \emptyset)\ \} \end{aligned}$$

The update for maybe transitions is simply the new obligations allowing previous obligations to be discarded. Maybe τ transitions, present in states with obligations, also discard previous obligations.

Definition 28 (Update Obl. Upon May. Trans. eTS)**.** Let ℓ be a label in $\Sigma \cup \{\tau\}$. We define $updateOblUMayT(\langle rtp, \Theta\rangle, \ell)$ as the set

$$\{\Theta' \mid \Theta' = newObl \ \wedge \ (\ell = \tau \implies \Theta \neq \emptyset) \wedge (updateTrig(rtp, \ell) \in L_T \implies newObl = L_M) \wedge (updateTrig(rtp, \ell) \notin L_T \implies newObl = \emptyset)\}$$

Note that in the above formalisation $updateOblUReqT$ and $updateOblUMayT$ return a set containing sets of obligations. This will ensure (in the next definition) that for every action that can consume the first action of multiple obligations, there will be one transition for each obligation. Thus, the non-determinism on a explained in the running example is achieved.

Definition 29 (Synthesis of MTS from eTS)**.** Let E be an eTS with trigger T, main chart M and alphabet Σ. The MTS synthesised from E is $W = (S, \Sigma, \Delta^r, \Delta^p, s_0)$ where

- $S = \{\langle\langle\alpha, \zeta\rangle, \Theta\rangle \cdot \langle\alpha, \zeta\rangle \in prefixes(L_T) \ \wedge \ \Theta \subseteq suffixes(L_M)\}$.
- $s_0 = \langle\langle\epsilon, \zeta^T\rangle, \Theta\rangle$ with $\Theta = L_M$ if $\langle\epsilon, \zeta^T\rangle \in L_T$ and $\Theta = \emptyset$ otherwise, where ζ^T is the state function derived from the set of fluents present in T.
- $\Delta^p = \{(\langle rtp, \Theta\rangle, \ell, \langle rtp', \Theta'\rangle) \mid updateTrig(rtp, \ell) = rtp' \wedge \Theta' \in (updateOblUMayT(\langle rtp, \Theta\rangle, \ell) \cup updateOblUReqT(\langle rtp, \Theta\rangle, \ell))\}$.
- $\Delta^r = \{(\langle rtp, \Theta\rangle, \ell, \langle rtp', \Theta'\rangle) \mid updateTrig(rtp, \ell) = rtp' \wedge \Theta' \in updateOblUReqT(\langle rtp, \Theta\rangle, \ell)\}$

It can be shown that the MTS W synthesised from an eTS E satisfies Property 1. This invariant is too weak to prove that W characterises through refinement all LTSs that satisfy E. However Property 1 together with the definition of the update functions imply that the synthesis procedure is correct and complete, thus W characterises the implementations that satisfy the scenario E.

Proposition 1. *If W is the MTS synthesised from E then Property 1 holds.*

Theorem 1 (Completeness and Correctness)**.** *Let $E = \diamond(T, M, \Sigma)$ be an eTS and W the MTS synthesised from E according to Definition 29, then, for every LTS I, $I@\Sigma \in \mathcal{I}[W]$ if and only if $I \models E$.*

4.1.3 Implementation

We have implemented the synthesis procedure defined above in the publicly available MTSA tool [20]. The implementation builds the MTS on the fly, starting from the initial state, and differs slightly from Definition 29 in that it produces an equivalent MTS but that has less transitions. This is achieved by using the maybe τ transitions to model other maybe behaviour: for each state, before adding a maybe t transition from s to s' the algorithm checks if a maybe τ and then

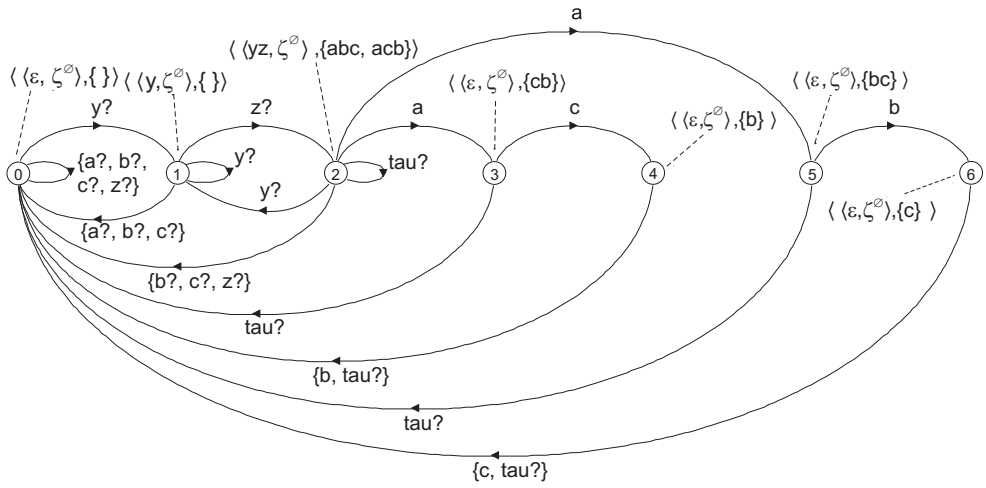


Figure 15: MTS synthesised from the eTS running example using MTSA

maybe t transition can be taken from s to s'. If so the maybe t transition is not added from s to s'. It is straightforward to show that such optimization is semantic preserving.

As an example consider Figure 15 which is the result of the optimised algorithm and Figure 12 which corresponds to that of Definition 29. It is simple to see that, for instance, outgoing maybe transitions from state 3 in Figure 12 (i.e. transitions on $a?$, $b?$, $c?$, $z?$ leading to 0 and $y?$ leading to 1) can be simulated by first performing $\tau?$ in Figure 15 from 3 to 0 and then the corresponding action label (e.g. $a?$ is simulated by $3 \xrightarrow{\tau}_{\mathrm{p}} 0 \xrightarrow{a}_{\mathrm{p}} 0$). On the other hand, the only maybe transition from state 3 in Figure 15 is a τ transition that is simulated by the τ transition from state 3 in the MTS in Figure 12.

Note that the MTSs depicted in Section 5 are those generated by MTSA.

4.1.4 Complexity

In this section we give an insight into the worst-case time complexity of the algorithm and the number of states of the synthesised MTS. Before starting the construction of the MTS L_T and L_M are calculated. The former is used to calculate the recognised trigger prefix and the later to be set as the set of obligations for states where the trigger hold. The algorithm makes a single traversal during which it adds states on the fly. For each state it may add transitions for each action ℓ in Σ. Each state of the generated MTS is created and then processed. Processing consists of calculating the recognised trigger prefix of the target state, $updateTrig(rtp, \ell)$, where rtp is the recognised trigger prefix of the source state and ℓ is the label of the transition being considered. Calculating the obligations of a successive state is trivial and done in constant time: obligations are set to L_M if the trigger is satisfied with the last transition and the empty set if there were no transitions before and the tail of a particular obligation if a required branch is being built. Therefore, the time complexity of the algorithm is $O(ComplexityCalcL_M + ComplexityCalcL_T + NumberOfStates\ |\Sigma|\ ComplexityUpdateTrig)$.

Each linearisation in L_M is calculated by taking a minimum location in the partial order and extending it according to the partial order till all the messages in M are used. In the worst case, when there is no particular order imposed to the messages in M, any permutation of those messages is in L_M. Therefore the number of linearisations in L_M is bounded by $|M|!$ (recall that being K an LPO or LPOC then $|K|$ is the size of any linearisation of K). The time complexity to build a single linearisation is bounded by $|M|$ when using efficient data structures to represent the partial order. Therefore the time complexity for calculating L_M ($ComplexityCalcL_M$) is bounded by $|M|!|M|$.

L_T contains the linearisations of T. But the linearisations of T are based on the linearisations of its associated LPO T_{LPO}. The number of linearisations of T_{LPO} ($|L_{T_{LPO}}|$) is bounded by $|T|!$. Building each linearisation takes a time bounded by $|T|$ and therefore the worst time complexity for calculating T_{LPO} is in $O(|T|!|T|)$. Recall that a linearisation of T is a linearisation of T_{LPO} and a state function. As there are F different valuations of the fluents, where F is the number of fluents affecting the trigger T, then the number of state functions are bounded by 2^F. Therefore the worst time complexity for building L_T ($ComplexityCalcL_T$) is the worst time taken to build the linearisations of T_{LPO} times 2^F: $O(2^F|T|!|T|)$.

Updating the trigger for a recognised trigger prefix $\langle \alpha, \zeta \rangle$ and action ℓ consists of checking if, after appending the last seen action to α, $\langle \alpha\ell, \zeta \rangle$ or any of $\alpha\ell$'s suffixes with the updated ζ function is a prefix of a word in L_T. In the worst case $\langle \alpha\ell, \zeta \rangle$ (where $\alpha\ell$ can not be longer than $|T|$) and every suffix of $\alpha\ell$ will have to be tested with each word in L_T which yields $O(ComplexityUpdateTrig) = O(|L_T||T|^2)$. This can be implemented more efficiently with a more time efficient data structure: The recognised trigger prefix keeps track of the suffixes that are a prefix of a word in L_T and keeps also a reference to that word in L_T so that, for a particular action, it is sufficient to try to extend those suffixes and look for the longest one that is a prefix of the trigger. Then, the complexity of updating the trigger ($ComplexityUpdateTrig$) is in $O(|L_T||T|) = O(2^F|L_{T_{LPO}}||T|) = O(2^F|T|!|T|)$.

We now calculate the size of the generated MTS. From the initial state new states are added creating a path monitoring the occurrence of the trigger. There is one path for each word in L_T and the length of the path is $|T|$. So there can be as many as $|L_T||T|$ states before a trigger is satisfied. As $|L_T|$ is bounded by $2^F|T|!$ then the number of states before the trigger is satisfied is bounded by $2^F|T|!|T|$. After the trigger holds there is a path for each word in the main chart going through $|M| - 1$ states, one for each prefix of that word. Therefore, if there is no nested triggering, then there will be $|L_M||M - 1|$ states after a trigger is satisfied. If a transition, along these paths where the main chart is being met, satisfies the trigger (i.e., there is a nested triggering) then a new state is added where, besides L_M, Θ also contains what is left of the obligation being met. As each nested trigger adds, in the worst case, one extra state there can be up to $|L_M|2|M-1|$ states. Finally, using that $|L_M|$ is bounded by $|M|!$, the number of states is bounded by $2^F|T|!|T| + 2|M|!|M|$. In practice the number of states is smaller than this as some valuations of the fluents are not possible after certain transitions. For example if an action ℓ sets a fluent f to *true* then there can not be a state where the recognised trigger prefix ends with ℓ and has a function where f is *false* . Also, in average, the possible linearisations of the main chart and trigger are much less than the worst case of $|M|!$ and $|T|!$ respectively. For instance in the case study presented in Section 5 all the scenarios have only one linearisation for the trigger and one for the main chart.

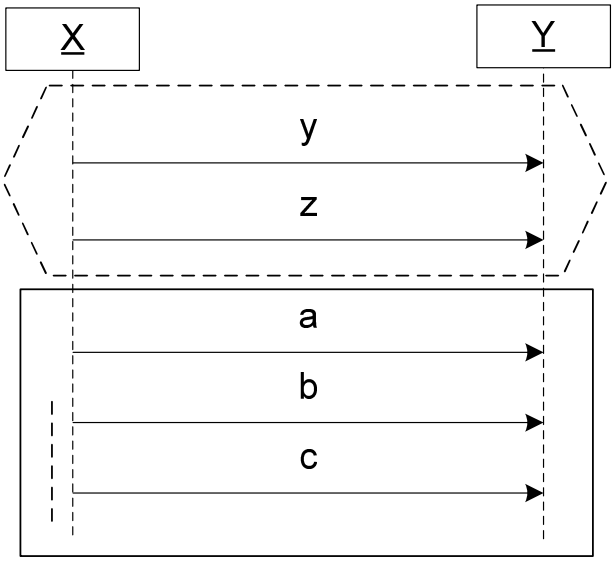


Figure 16: The uTS used as the synthesis algorithm's running example

Summing up, the worst-case time complexity of the algorithm is $O(|M|!|M| + 2^F|T|!|T| + ((2^F|T|!|T| + 2|M|!|M|)|\Sigma|(2^F|T|!|T|)))$. Let $m = |M|!|M|$ and $t = 2^F|T|!|T|$. Then we can rewrite the formula as $O(m+t+((t+2m)|\Sigma|t)) = O(m+t+|\Sigma|t^2+2|\Sigma|tm) = O(|\Sigma|t^2+|\Sigma|tm) = O(|\Sigma|(t^2+tm)) = O(|\Sigma|((2^F|T|!|T|)^2 + (2^F|T|!|T||M|!|M|)))$.

Scenarios only have a few messages and fluents affecting the scenario's trigger so the number of variables affecting the complexity are generally small in practice.

The algorithm for merging two MTSs starts by computing a common refinement and then successively builds a more abstract MTS. Merging is exponential on the degree of non-determinism of the common refinement from which it first starts the abstraction process [16, 18]. The degree of non-determinism of a model at a given state and label is equal to the number of outgoing transitions with that label minus one. The degree of non-determinism of an MTS is the sum of the degree of non-determinism for every state and label. The case study presented in Section 5 confirmed that the time taken for synthesising models was negligible compared to the time taken for merging those models. It took, for each scenario, less than a second to synthesise each MTS. On the other hand it took a couple of minutes to merge some of the largest models.

4.2 Synthesis from uTS

4.2.1 Running example

Let us now consider the uTS in Figure 16. Let T be the trigger and M the main chart. Then $L_T = \{\langle yz, \zeta^{\emptyset}\rangle\}$, $L_M = \{abc, acb\}$ and the alphabet is $\Sigma = \{a, b, c, y, z\}$. Note that this scenario is identical to the one used in the previous section except that we now take a universal, rather than existential, interpretation.

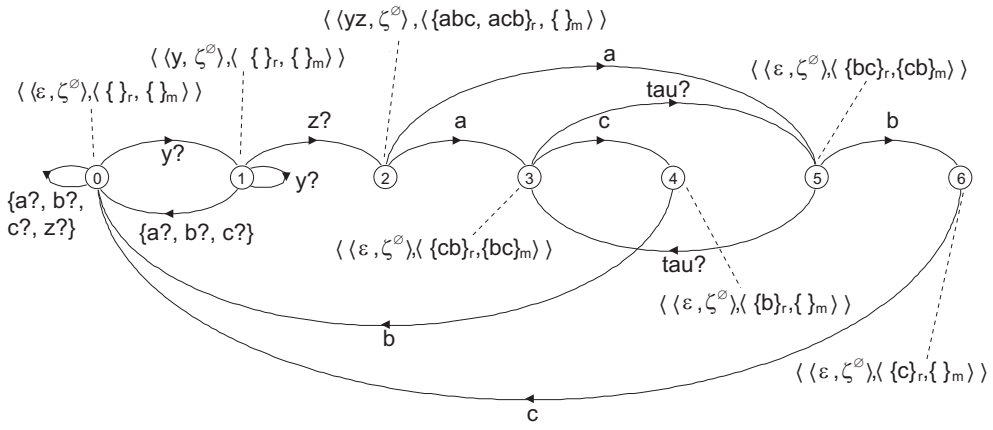


Figure 17: MTS synthesised from the uTS running example with states annotated with the state's structure

As with the synthesised model from an eTS, the MTS synthesised from an uTS has to keep track of the prefix of the trigger that has been recognised and enforce mandatory behaviour once the trigger has occurred. The difference is that the MTS for a uTS should not allow behaviour not described in the main chart (L_M). A naive approach to synthesis would be to reuse the eTS synthesis algorithm and simply remove maybe transitions from states with obligations. Such an approach is incorrect: Consider the MTS in Figure 12 without the maybe transitions originating from states 3 through 6. The LTS in Figure 18 would not be a refinement of the MTS yet the LTS does satisfy the uTS.

The MTS depicted in Figure 17 characterises through refinement the LTSs that satisfy the uTS of Figure 16. States 0 and 1, and their outgoing transitions are identical to those in the MTS synthesised for the existential version of the scenario (see Figure 12) because they are intended to fulfil the same purpose: monitor the occurrence of the trigger and guarantee that if the trigger is satisfied the resulting state is 2. The MTSs for the universal and existential TS are also similar in that they have a non-deterministic choice for a on state 2. This is to avoid, as explained for eTSs, losing the implementation in Figure 14 which satisfies the uTS.

Where the MTS for the universal scenario differs is in the maybe transitions from states with obligations. For uTS these transitions should only allow behaviour described in the main chart. The MTS in Figure 17 only has two maybe τ transitions: from state 3 to 5 and back. These transitions are needed to allow LTS implementations that provide the behaviour of the main chart in a deterministic fashion. Consider the LTS in Figure 14 but in which states 3 and 5 have been joined (i.e. state 2 goes to 3/5 via a and then there is a choice on c and b to go to states 4 and 6 respectively). Such an LTS satisfies the uTS but would not be an implementation of the MTS in Figure 17 without its τ transitions as the latter requires committing early to whether *abc* or *acb* will be provided while the former delays the choice until after a has occurred. Note that in the model synthesised from a eTS those maybe τ transitions from states with obligations also exist but they do not necessarily go to states with obligations (unless the trigger holds) as the implementations satisfying the scenario are not required to show the main chart's behaviour in

every run.

So now we have two kind of obligations: required obligations and maybe obligations. Maybe obligations can appear along a required path, that is, while the main chart is being met. The maybe obligations represent the paths that should not be forbidden in the implementations.

4.2.2 Synthesis

The synthesis strategy for uTS is similar to that of eTS. States are still encoded with a structure with two parts, the recognised trigger prefix up to that state and the state's pending obligations. However, the notion of obligation changes to conform to the semantics of universal: First, the representation of obligations at states of the synthesised MTS changes, and second, the way obligations are updated once a transition is traversed differs.

To describe the obligations of a state we now use two sets of words: required obligations and maybe obligations. Required obligations are words for which required paths from the state are expected to exist. Maybe obligations are words for which paths from the state could exist. Consider for instance state 2 in Figure 17 which has two words in the required obligations set (*abc* and *acb*) and no words in the maybe obligations set. This is consistent with the fact that from state 2 required paths for *abc* and *acb* exist.

Consider state 3 in the same MTS, this state has only one required obligation which is *cb* representing the fact that the action *a* that is required by the uTS has occurred and *cb* remains. There is no need to have *bc* as an obligation as state 5 in the MTS guarantees such path from state 2. However, *bc* should not be prohibited in 3, hence this state also has one maybe obligation. This maybe obligation is fulfilled by the possible path from 3 through 5, 6, 0. If Θ represents the obligations of a state, then we will refer to the required and maybe obligations as $\Theta.r$ and $\Theta.m$ respectively.

We now explain the invariant that holds for all states $\langle\langle\alpha, \zeta\rangle, \Theta\rangle$ of an MTS synthesised from a uTS (Property 2): Firstly, every trace π of the MTS leads to a state with a recognised trigger prefix obtained as the significant suffix of π and its corresponding state function ζ (item 1 of Property 2). From every state, the outgoing paths of required transitions are exactly those in the state's required obligations (implied by item 2 of Property 2) and that any word w in the state's maybe obligations can be replayed from that state (item 3). A state with no required obligations has outgoing transitions on every action of the alphabet (item 4) i.e. if there are no obligations any action should be possible.

Property 2 (Invariant). Let W be an MTS synthesised from an uTS with trigger T, main chart M and alphabet Σ. For all trace $\pi \in (\Sigma \cup \{\tau\})^*$ such that $W \xrightarrow{\pi}_{\mathrm{p}} W_s$ with $s = \langle rtp, \Theta\rangle$ then:

1. $sigSuf(\langle \pi|_\Sigma, \zeta^T\rangle) = rtp$, where ζ^T is the state function derived from the fluents present in T.
2. $\forall \ell \in \Sigma \cdot W_s \xrightarrow{\ell}_{\mathrm{r}} \Leftrightarrow \exists\, \theta_r \cdot \ell\theta_r \in \Theta.r$
3. $\forall \theta_m \in \Theta.m \cdot W_s \overset{\theta_m}{\Longrightarrow}_{\mathrm{p}}$
4. $\Theta.r = \emptyset \Rightarrow (\forall \ell \in \Sigma \cdot W_s \xrightarrow{\ell}_{\mathrm{p}})$

Note that the invariant of an MTS synthesised from uTS is similar to the one for MTS synthesised from eTS. The difference is that, besides having required paths if and only if the paths corresponds to required obligations (or plain obligations in the case of eTS), in the case of uTS the presence of possible paths corresponding to maybe obligations have to be guaranteed (item 3). The last difference is that in the case of eTS any word over the alphabet is a possible trace from any state, however, because of the semantics of uTS, the only states in the synthesised MTS that can allow any possible transition are the ones where no required obligation is present (item 4).

As with eTS, the MTS synthesis procedure adds a transition labelled ℓ between two states if and only if updating the recognised trigger prefix and obligations of the transition's source with ℓ is compatible with the recognised trigger prefix and obligations of the transition's target. The update of recognised trigger prefixes remains as for eTS.

The update of the obligations of state $\langle rtp, \Theta \rangle$ after a required transition labelled ℓ (see Definition 30) is based on the following criteria. The update is allowed only if there is a required obligation starting with ℓ ($\exists w \cdot \ell w \in \Theta.r$) and the resulting obligations depend on whether the occurrence of ℓ satisfies the trigger given that the current recognised trigger prefix is rtp. If the trigger is satisfied, then the required paths from the resulting state must be L_M ($\Theta'.r = L_M$). As with state 2 in Figure 17, there is no need to have maybe obligations ($\Theta'.m = \emptyset$) as all potential maybe obligations are already in $\Theta'.r$. If the trigger is not satisfied and the occurrence of ℓ fulfils an entire obligation ($w = \epsilon$), then there are no obligations of any kind in the next state $\Theta'.r = \Theta'.m = \emptyset$. Such is the case of the obligations after c (resp. b) from state 6 (resp. 4). Finally, if the trigger is not satisfied and ℓ contributes to fulfilling an obligation but there are remaining required actions ($w \neq \epsilon$), then what is left becomes the required obligation ($\Theta'.r = \{w\}$) and all other required and maybe obligations of the original state to which ℓ contributes become part of the maybe obligations $\Theta'.m = \{w' \neq w \mid \ell w' \in (\Theta.m \cup \Theta.r)\}$. This is the case of the obligations of state 3, for instance, after the occurrence of a from state 2.

Definition 30 (Update Obl. Upon a Req. Trans. uTS)**.** Let ℓ be a label in Σ. We define $updateOblUReqT(\langle rtp, \Theta \rangle, \ell)$, where $b = updateTrig(rtp, \ell) \in L_T$, as the set

$$\{\ \Theta' \mid \exists w \cdot \ell w \in \Theta.r \wedge (b \implies \Theta'.r = L_M \ \wedge\ \Theta'.m = \emptyset) \wedge (\neg b \ \wedge\ w = \epsilon \implies \Theta'.r = \Theta'.m = \emptyset) \wedge (\neg b \wedge\ w \neq \epsilon \implies \Theta'.r = \{w\} \wedge \Theta'.m = \{\ w' \neq w \mid \ell w' \in (\Theta.m \cup \Theta.r)\ \})\ \}$$

The update of the obligations of state $\langle rtp, \Theta \rangle$ after a maybe transition labelled ℓ (see Definition 31) is based on the following criteria: A maybe transition is always allowed on states with no obligations ($\Theta.r = \emptyset$) and the resulting required obligations depend on whether the occurrence of ℓ satisfies the trigger ($updateTrig(rtp, \ell) \in L_M$). The maybe z transition from state 1 is an example of the former when the trigger is satisfied, while the maybe y transition from the same state is an example for when the trigger is not satisfied. When there are both required and maybe obligations then the only maybe transitions that are allowed are τ transitions. The new required and maybe obligations are the result of swapping the old required obligation with one of the maybe obligations ($\exists w \in \Theta.m \cdot \Theta'.r = \{w\} \wedge \Theta'.m = (\Theta.m \cup \Theta.r) \setminus \{w\}$). This is the case of the τ transitions to and from states 3 and 5. Finally, when there are required but no maybe obligations, then no maybe transitions are allowed. Such is the case of state 2.

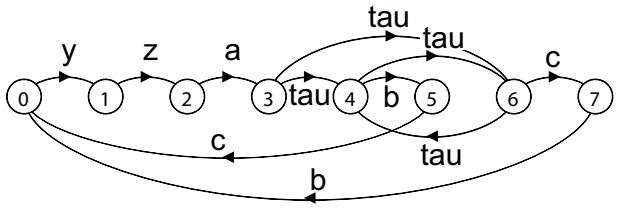


Figure 18: An implementation of the MTS in Figure 17

Definition 31 (Update Obl. Upon May. Trans. uTS)**.** Let ℓ be a label in $\Sigma \cup \{\tau\}$. We define $updateOblUMayT(\langle rtp, \Theta\rangle, \ell)$, where $b = updateTrig(rtp, \ell) \in L_M$, as the set

$$\{\ \Theta' \mid (\Theta.r = \emptyset\ \wedge b \Longrightarrow \ell \neq \tau \wedge\ \Theta'.r = L_M\ \wedge\ \Theta'.m = \emptyset) \wedge$$
$$(\Theta.r = \emptyset\ \wedge \neg b \Longrightarrow \ell \neq \tau \wedge\ \Theta'.r = \Theta'.m = \emptyset) \wedge$$
$$(\Theta.r \neq \emptyset \Longrightarrow \ell = \tau \wedge\ \exists w \in \Theta.m\ \cdot\ (\Theta'.r = \{w\} \wedge \Theta'.m = (\Theta.m \cup \Theta.r) \setminus \{w\}))\}$$

Finally, we provide the construction of MTS from uTS. The resulting MTS is guaranteed to satisfy the invariant formalised in Property 2.

Definition 32 (Synthesis of MTS from uTS)**.** Let U be an uTS with trigger T, main chart M and alphabet Σ. The MTS synthesised from U is $W = (S, \Sigma, \Delta^r, \Delta^p, s_0)$ where

- $S = \{\langle\langle\alpha, \zeta\rangle, \Theta\rangle \cdot \langle\alpha, \zeta\rangle \in prefixes(L_T)\ \wedge\ \Theta.r, \Theta.m \subseteq suffixes(L_M)\}$.
- $s_0 = \langle\langle\epsilon, \zeta^T\rangle, \Theta\rangle$ with $\Theta.m = \emptyset$, and $\Theta.r = L_M$ if $\langle\epsilon, \zeta^T\rangle \in L_T$ and $\Theta.r = \emptyset$ otherwise, where ζ^T is the state function derived from the set of fluents present in T.
- $\Delta^p = \{(\langle rtp, \Theta\rangle, \ell, \langle rtp', \Theta'\rangle) \mid updateTrig(rtp, \ell) = rtp' \wedge \Theta' \in (updateOblUMayT(\langle rtp, \Theta\rangle, \ell) \cup updateOblUReqT(\langle rtp, \Theta\rangle, \ell))\}$.
- $\Delta^r = \{(\langle rtp, \Theta\rangle, \ell, \langle rtp', \Theta'\rangle) \mid updateTrig(rtp, \ell) = rtp' \wedge \Theta' \in updateOblUReqT(\langle rtp, \Theta\rangle, \ell)\}$

Proposition 2. *If W is an MTS synthesised from a uTS U then Property 2 holds.*

As with eTS, the invariant in Property 2 is too weak to prove that W characterises through refinement all LTSs that satisfy U. However Property 2 together with the definition of the update functions imply that the synthesis procedure for uTS is correct and complete, thus W characterises the implementations that satisfy the scenario U.

Theorem 2 (Completeness and Correctness)**.** *Let $U = \Box(T, M, \Sigma)$ be an uTS and W the MTS synthesised from U according to Definition 32, then, for every LTS I, $I@\Sigma \in \mathcal{I}[W]$ if and only if $I \models U$.*

The examples presented in the previous two sections dealt with triggers with no conditions. As a final example consider the uTS in Figure 8 with trigger T and let $\Phi_2 = \phi_2$ where ϕ_2 is the fluent $\langle\{b\}, \{c\}, \top\rangle$ initially *true* and set to *false* with c and *true* with b. Figure 19 shows

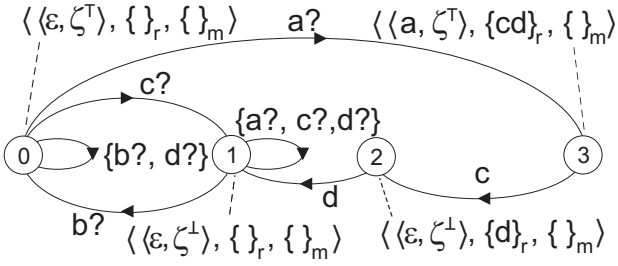


Figure 19: MTS synthesised from the uTS in Figure 8 with states annotated with the state's structure

the MTS synthesised from the aforementioned scenario where $\zeta^{\top}$ and $\zeta^{\perp}$ are the state functions that evaluate ϕ_2 to *true* and *false* respectively. The MTS is synthesised starting with the state function derived from ϕ_2, the only fluent in the trigger, which equals $\zeta^{\top}$. Then the initial state's recognised trigger prefix is $\langle \epsilon, \zeta^{\top} \rangle$. Note how, from the initial state, the occurrence of c leads to state 1 were the state function is updated to $\zeta^{\perp}$. The occurrence of a at state 1 does not trigger the scenario because the condition does not hold. The recognised trigger prefix of the state reached from state 1 through an a transition is: $updateTrig(\langle \epsilon, \zeta^{\perp} \rangle, a) = sigSuf(\langle a, \zeta_a^{\perp} \rangle)$. The valuation of the fluent does not change with a so $\zeta_a^{\perp} = \zeta^{\perp}$ and, as $\langle a, \zeta^{\perp} \rangle$ is not a prefix of L_T, then $sigSuf(\langle a, \zeta_a^{\perp} \rangle) = \langle \epsilon, \zeta^{\perp} \rangle$. That explains the self transition a at state 1. Similarly, a b transition from state 1 changes the valuation of the fluent thus leading to state 0. An a transition from the initial state triggers the scenario leading to state 3 and from there there is a required branch with the main chart's behaviour. From state 3 to state 2 the state function changes because the valuation of ϕ_2 becomes *false* with the occurrence of c. The final d transition of the required branch ends up in state 1 because d does not change the valuation of the fluent to *true* .

4.2.3 Implementation

We have implemented the synthesis procedure defined above in the publicly available MTSA tool [20]. As with the eTS synthesis implementation, it builds the MTS on-the-fly from the initial state. Note that the MTSs depicted in Section 5 correspond to those generated by MTSA.

4.2.4 Complexity

The space complexity of the uTS synthesis algorithm is the same as that for eTS synthesis, the same arguments as in 4.1.4 apply.

The time complexity is pretty much the same only that the obligations of the target state are not longer calculated in constant time. While creating a required branch for a particular obligation $t\theta$, where $t \in \Sigma$, all the other required obligations of the origin state have to be traversed to checked if they start with t. In that case they will be part of maybe obligations of the target state. The time complexity of this operation is bounded by $|L_M|$.

So calculating the target state after a transition is now $O(NumberOfStates\ |\Sigma|\ ComplexityUpdateTrig$

ComplexityTargetObligations). Where O(*NumberOfStates*), and O(*ComplexityUpdateTrig*) are the same as for eTS and O(*ComplexityTargetObligations*) = L_M.

Finally the time complexity of the uTS synthesis algorithm is O(*ComplexityCalc*L_M+*ComplexityCalc*L_T+ *NumberOfStates* |Σ| *ComplexityUpdateTrig ComplexityTargetObligations*).

5 Validation

In this section, we report on our experience validating the results described in previous sections. We report on a case study aimed at using triggered scenarios, MTS synthesis and MTS analysis to iteratively and incrementally elaborate behaviour models.

5.1 Tool Support

Support for writing triggered scenarios and for synthesising MTS from them has been incorporated into the prototype Modal Transition System Analyser (MTSA) [29, 20]. MTSA supports various forms of constructing and analysing MTS models. Models can be described using traditional process algebra operators such as sequential and parallel composition, and hiding, as well as the MTS merge operator [18]. In addition to synthesis from triggered scenarios, MTSA supports synthesis from non-triggered existential scenarios and from safety properties expressed in linear temporal logic of fluents (FLTL) [25].

The MTSA tool supports analysis of MTS models through standard model-based validation techniques such as inspection (both of the textual and graphical representation of the MTS), animation, hiding, minimisation, and model checking. The latter includes checking an MTS for deadlock freedom and against FLTL properties, in addition to comparing models for consistency and refinement. Validation of the approach described in this paper was performed using MTSA.

5.2 Methodology

Case studies were conducted by iterating a synthesise-analyse-elicit cycle. In the *synthesis* phase, an MTS is automatically constructed from known properties and scenarios. In the *analysis* phase, the synthesised MTS is analysed, using MTSA, via inspection, animation, model checking and model slicing (action hiding plus minimisation). In the *elicitation* phase, questions prompted during the analysis are answered based on the domain knowledge available and modelled in order to return to the *synthesis* phase with a more elaborated specification. The stopping criteria for the iterations is the production of a fully specified behaviour model in the form of a LTS that is a valid description of the system being modelled.

Before discussing the case studies, a few words on the analysis phase. During the analysis phase of each of the case study's iterations we used a combination of these techniques that are common to behaviour model analysis in general (inspection, animation, model checking and model slicing). The key driving force in the analysis phase of each iteration is to provide insights into the underspecified behaviour, captured explicitly as maybe transitions. Hence, much of

the analysis consists of identifying reachable maybe transitions. In the elicitation phase, what-if questions are constructed by traces leading to these maybe transitions to elicit if the maybe behaviour should be refined into required or prohibited behaviour.

Another key driving force during analysis is to consider the two bounds captured by the synthesised MTS: the behaviour proscribed by the MTS and the behaviour required by the MTS. The *pessimistic* and *optimistic* ([19]) views of an MTS naturally support this analysis. The pessimistic implementation of an MTS is the implementation where all maybe behaviour in the MTS is forbidden. In other words, only the behaviour required by the specification is present in the implementation. Similarly, the optimistic implementation is the implementation where all maybe behaviour has been converted to required behaviour. In other words, any behaviour not exhibited in the optimistic implementation is behaviour proscribed by the specification.

In the description of the case studies, rather than focusing on how analysis was performed, we focus on the questions prompted by the analysis. When considered of interest, we do explicitly point out a specific technique that led to a relevant question in the elicitation phase. We discuss the analysis phase in more detail in the conclusion of this section.

5.3 Philips Television Set Configuration

This section reports on a case study of an industrial protocol for a product family of Philips television sets [30]. The TV product family can include multiple tuners and multiple video output devices that can be configured to display several signals in different configurations. The protocol is concerned with controlling the signal path in a TV to avoid visual artifacts appearing on video outputs when a tuner is changing frequency.

The setup for this case studies was as follows. In addition to the available documentation of the protocol, we were provided with a prototype in which various TV architectures could be configured. The prototype supports exploration of the behaviour of the tuning protocol for each architecture of the system. It could therefore be used as a replacement for a domain expert in the *elicitation* phase. Observed behaviour into the prototype was initially encoded as existential and universal triggered scenarios, and an MTS was synthesised. This was analysed with the view of posing questions regarding the *maybe* behaviour of the MTS (should certain *maybe* behaviour exhibited by the MTS be mandatory or proscribed?) which were answered by replaying specific situations in the prototype and observing its response. Exercising the prototype to validate the MTS model and answer questions regarding its *maybe* behaviour generated further observations of the protocol's behaviour that were encoded in new scenarios and properties leading to the next iteration of the synthesise-analyse-elicit cycle.

In the case study reported below, the architecture of the TV is fixed to having two tuners and one video output. The two tuners are connected with the single video output through a switching device which displays the signal of the active tuner. The active tuner can be changed by a user interacting with a switching device. The user can also change the frequency of either tuner. The protocol coordinates the tuners, video and switch devices in order to ensure that the video does not produce an output while the signal is being changed. This first example focuses on the behaviour of the protocol with respect to changes in the tuning frequencies. The second focuses on the behaviour resulting from switching active tuners.

5.3.1 Tuning

As explained above, we setup the prototype as a TV with two tuners (*t1*, *t2*), a switch (*s*) and a video output (*v*) with the active tuner initially being *t1*.

Firstly, we explored the basic tuning behaviour of the TV by changing the frequency of the active tuner: Once a *tune* command is sent to the tuner, it stores the new frequency and requests the switch to drop the signal corresponding to the frequency being displayed up to that moment (*dropReq*). The switch forwards the drop signal request to the video output and then sends an acknowledgement (*dropReqAck*) back to the tuner to confirm that the video signal has been dropped and hence a blank screen is being displayed. Finally the tuner changes the frequency of the signal being transmitted and requests the switch to restore the image on the video output (*restore*). The switch forwards the request and the video unblanks the screen and outputs the signal which corresponds to the new frequency.

The observed behaviour described in the previous paragraph is modelled in the eTS *E_Tuning_t1_Active_t1* of Figure 20. Fluent *Active_t1* represents the status of the tuner *t1*: It is initially *true* and becomes *false* when tune *t2* is activated, and *true* when *t1* is activated. For the sake of simplicity, the actual change of frequency is not modelled.

The rationale for selecting the particular eTS of Figure 20 was based on our understanding of the general description available for the protocol which explains that the system *reacts* to changes in the tuned frequency. Thus the eTS trigger is a tune command while the tuner *t1* is active. An alternative, weaker generalization would have been to move some more messages from the main chart into the trigger of the existential scenario, thus introducing a stronger antecedent (the trigger) and hence more restricted conditions for requiring the consequent (the main chart).

A stronger generalization of eTS *E_Tuning_t1_Active_t1* would have been to choose a universal scenario instead of an existential one to encode the observed behaviour. Such an encoding would imply that the main chart is the only behaviour that can be observed when a tuner is retuned. Clearly, at such an early stage of behaviour exploration it is unknown if behaviour other than that of the main chart can occur after the trigger. In fact, subsequently, it becomes clear that a universal scenario would have been incorrect as it is possible to retune in the middle of the behaviour described by the main chart of eTS *E_Tuning_t1_Active_t1*.

The MTS synthesised from eTS *E_Tuning_t1_Active_t1* is quite small (see Figure 21), and so inspection of the graphical representation is feasible: Note that in state 2, it is guaranteed that the trigger holds and that the trace *t1_newValue t1_dropReq s_dropReq s_dropReqAck_t1 t1_restore s_restore* is required from that same state. Hence, the required behaviour will be present in every implementation satisfying the trigger. In addition note that in every state from 2 to 7 there is an outgoing sequence of maybe transitions τ, *t1_tune* leading back to state 2, the occurrence of which restarts the tuning protocol.

The latter observation prompted two questions: Should tuning be allowed once the protocol is engaged? and if so, would the protocol have to restart or is there some notion of current state that is preserved for dealing with a new tune action? These questions were prompted by inspecting the maybe behaviour of Figure 21 which is the result of an existential scenario. An MTS synthesised from a universal version of *E_Tuning_t1_Active_t1* would not have included this maybe behaviour as it would have already proscribed the occurrence of a nested tune.

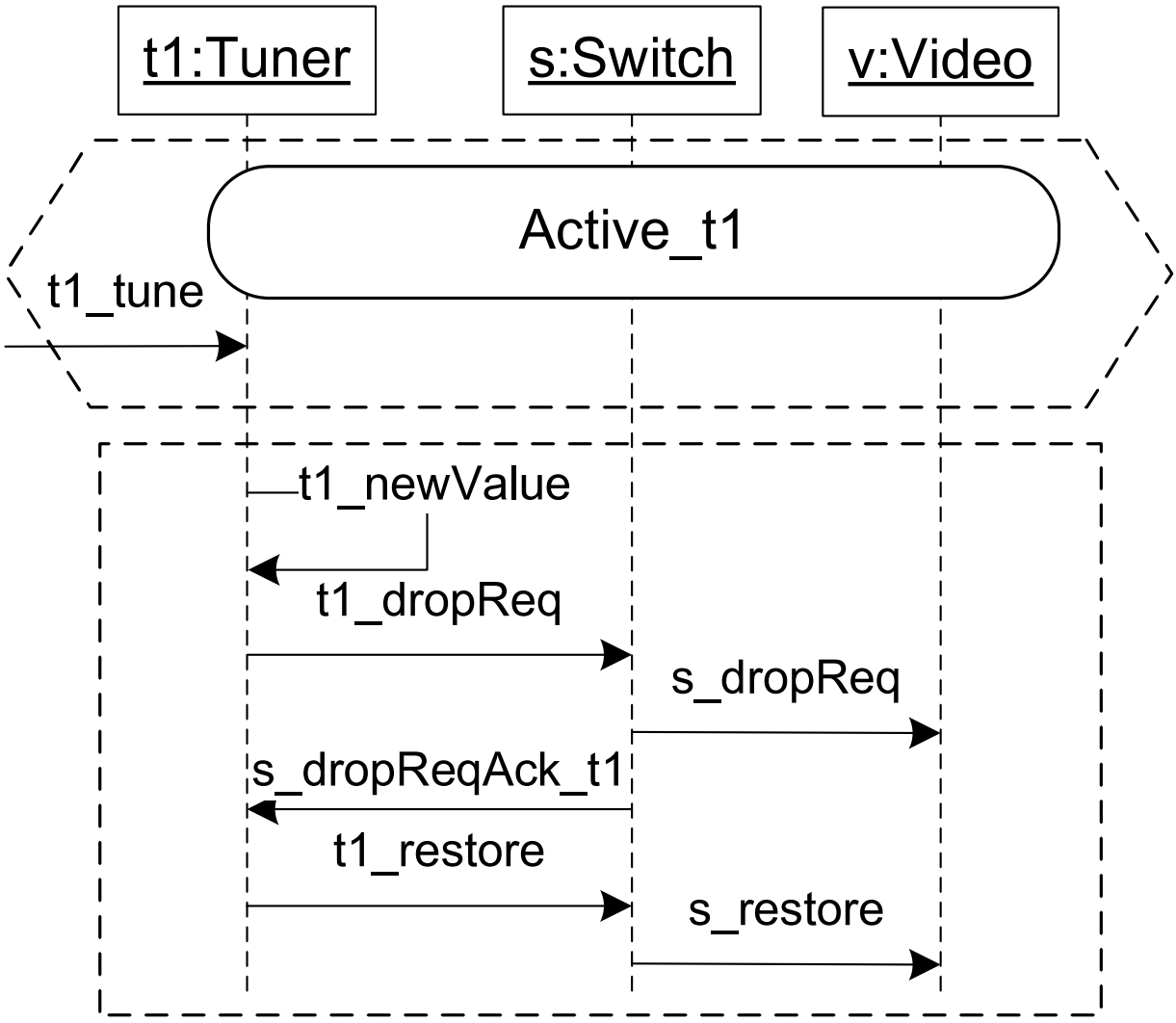
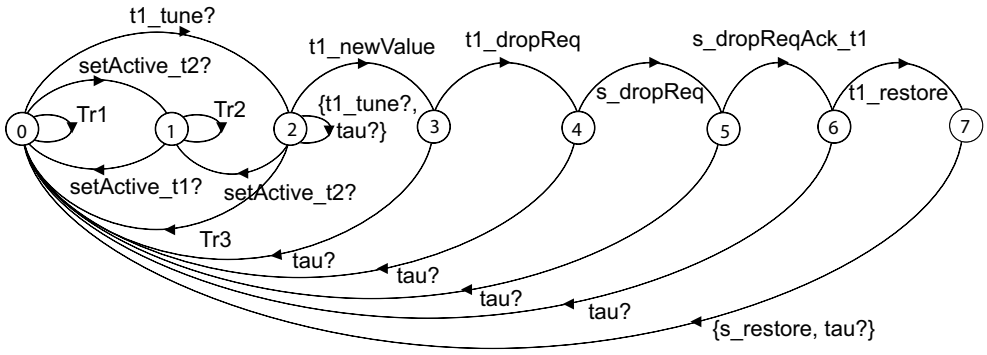


Figure 20: *E_Tuning_t1_Active_t1*



Tr1 = { s_dropReq?, s_dropReqAck_t1?, s_restore?, setActive_t1?, t1_dropReq?, t1_newValue?, t1_restore? }
Tr2 = { s_dropReq?, s_dropReqAck_t1?, s_restore?, setActive_t2?, t1_dropReq?, t1_newValue?, t1_restore?, t1_tune? }
Tr3 = { s_dropReq?, s_dropReqAck_t1?, s_restore?, setActive_t1?, t1_dropReq?, t1_restore? }

Figure 21: Synthesised MTS from the eTS in Figure 20

To answer these questions we replayed the trigger on the prototype and then attempted to tune before the protocol described in the main chart of *E_Tuning_t1_Active_t1* finished. Indeed, it was possible to retune but the nested occurrence did not restart the protocol: Once a nested tune occurs, as the signal on the video output is being dropped or has already been dropped, a further change in the signal's frequency will not produce any undesired video artefacts, hence the signal can be changed safely and no additional communication is required.

The behaviour regarding nested tuning is captured by strengthening the trigger of the original *E_Tuning_t1_Active_t1* scenario with a *Tuning_t1* fluent that is initially *false* , becomes *true* with *tune_t1* and *false* when the protocol finishes with *s_restore* or is aborted by the activation of any of the tuners with the actions *set_Active_t1* or *set_Active_t2* (see Figure 22). In addition, a second existential scenario named *E_NestedTuning_t1_Active_t1* (Figure 23) is added reflecting the fact that a nested tune will only trigger the storing of the new frequency value instead of dropping and restoring the signal on the video output.

The *restricts* set in the new scenario (*E_NestedTuning_t1_Active_t1*) is necessary to avoid the protocol being restarted after the nested tune. This alphabet extension forces the occurrence of *t1_new_value* before any other message of the protocol.

A new MTS can be constructed by merging the MTS synthesised from scenarios *E_Tuning_t1_Active_t1* and *E_NestedTuning_t1_Active_t1*, resulting in MTS *It2* shown in Figure 24.

Analysis of the maybe behaviour of *It2*, lead to the following finding: *if* a nested tune occurs, leading to state3, it triggers the store of the new frequency value (in $Tr6$). However, when can a nested tune occur? At any point? Which of the maybe transitions for these nested tunes should be required transitions? By exercising the prototype it becomes clear that a nested tune is not always allowed. In fact, once the protocol is engaged, it is only possible to retune on two occasions. The first one is when the switch has sent a drop request and the tuner is waiting the drop acknowledge from the switch. The second time is right after the drop acknowledge was received by the tuner and before the tuner sends the restore request. We call these two sections of the protocol *store-only sections*. A tune within those sections will not restart the protocol but instead only store the

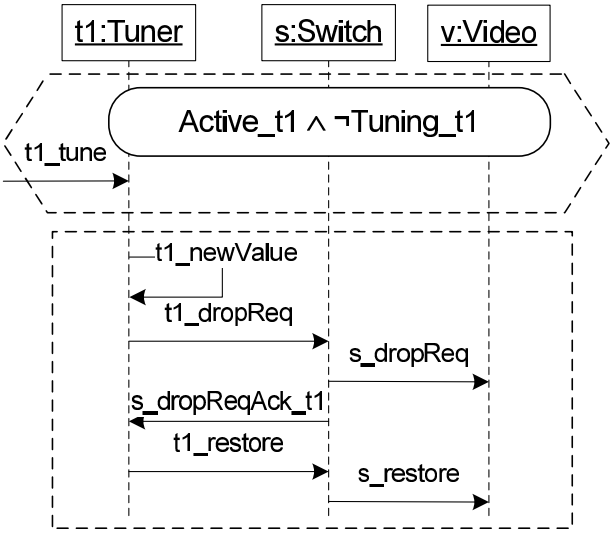


Figure 22: E_Tuning_t1_Active_t1 modified with a stronger condition

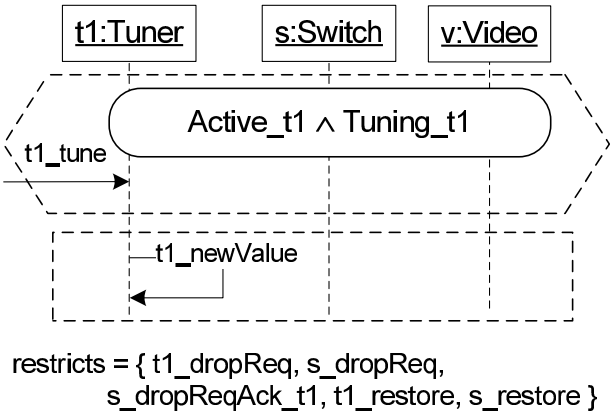


Figure 23: E_NestedTuning_t1_Active_t1

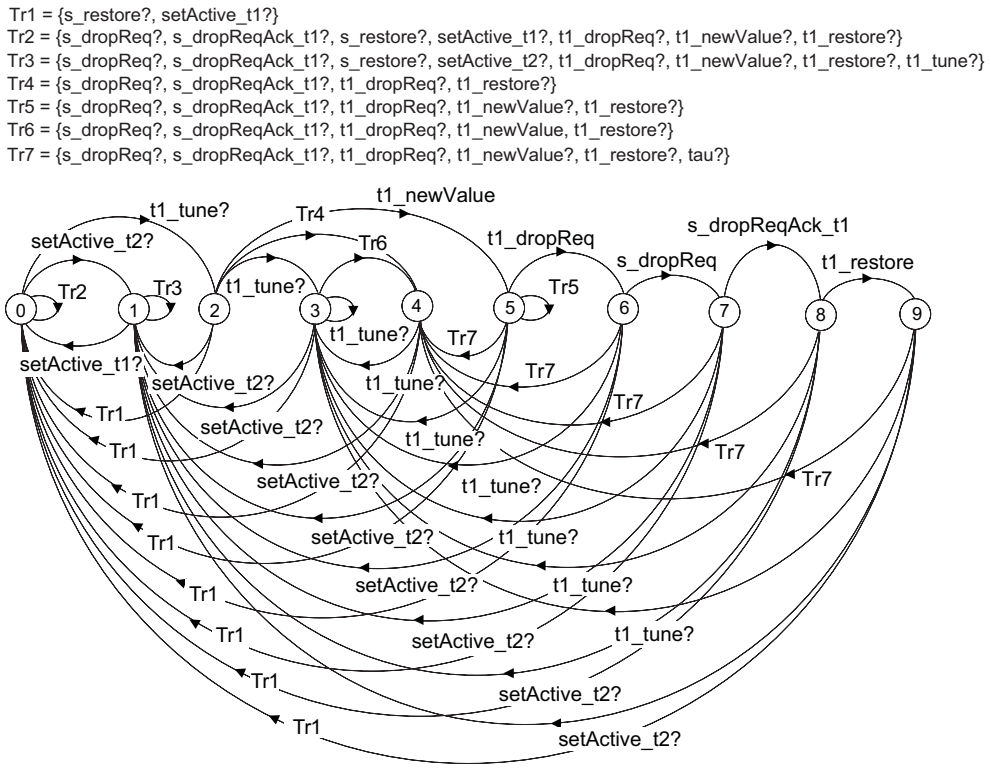


Figure 24: Resulting model after the second iteration: *It2*

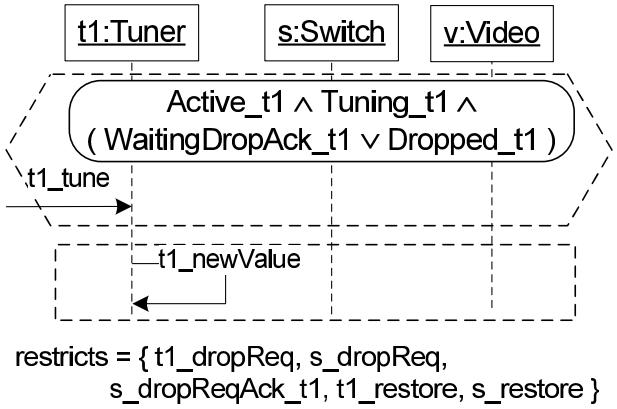


Figure 25: E_NestedTuning_t1_Active_t1 modified with a stronger condition

new frequency value. A nested tune outside that sections is not allowed in the prototype.

Based on the above observations, a strengthened version of *E_NestedTuning_t1_Active_t1* was produced (see Figure 25). This new eTS includes the fluents *WaitingDropAck_t*1: initially *false* , *true* with *t1_dropReq* and *false* with *s_dropReqAck_t1*; and *Dropped_t1*: initially *false* and *true* with *s_dropReqAck_t1* and *false* with *t*1*_restore* to signal the store-only sections. Furthermore, to reflect the fact that a nested tune is forbidden other than in the store-only sections we specify a precondition for the action *tune*:
Pre(tune_t1) = $\neg$*Tuning_t1* $\vee$ (*WaitingDropAck_t1* $\vee$ *Dropped_t1*)
which can be formalised using the FLTL property *Pre_tune_t1* = $\Box$(*Tuning_t1* $\wedge$ $\neg$(*WaitingDropAck_t1* $\vee$ *Dropped_t1*) $\rightarrow$ $\neg X$ *t1_tune*). The propositions appearing in the formula are previously defined fluents except for *t1_tune* which is an implicit fluent ([25]) derived from the action *t1_tune* such that it is initially *false* and becomes *true* only with that action and *false* with any other.

A new MTS *It3* can be constructed merging the MTS synthesised from the strengthened versions of *E_Tuning_t1_Active_t1* and *E_NestedTuning_t1_Active_t1*, and property *Pre_tune_t1*.

So far we have not specified under which conditions tuning *must* be allowed. Instead, we have elicited the behaviour of the protocol that is triggered by the occurrence of tuning.

For instance, in *It2*, from the initial state, a maybe *t1_tune* transition appears when, in fact, from exercising the prototype we know that this behaviour is present. Hence, a rule for introducing a required *t1_tune* transition from the initial state is needed. Generalizing, a new eTS called *E_TuneAllowed_t1* (Figure 26) is added to the specification, synthesised and merged with the analysed one. The resulting MTS (*It4*) is not shown due to its size. Instead we show its *pessimistic* implementation (Figure 27).

Analysis indicates a liveness problem. In Figure 27 states 5 and 6 form a strongly connected component where no *s_dropReqAck_t1* transition appears. In the same way, states 8 and 9 form another strongly connected component where no *t1_restore* transitions appear. This is a clear indication of a problem, as the prototype does not exhibit such behaviour: dropping and restoring occurs even if a second tune is invoked. If a trace leading to state 6 is animated in the synthesised MTS instead of its pessimistic version, it can be observed that in the state that is reached, there is

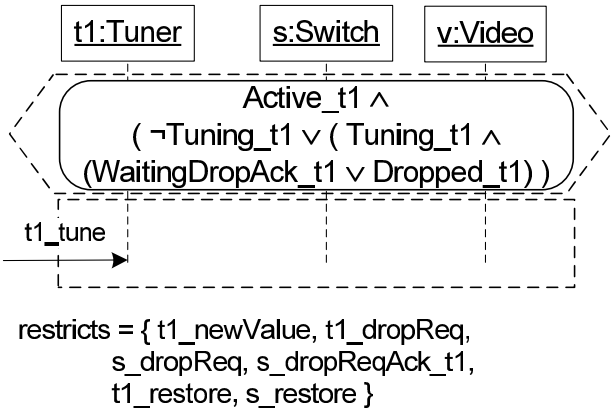


Figure 26: E_TuneAllowed_t1

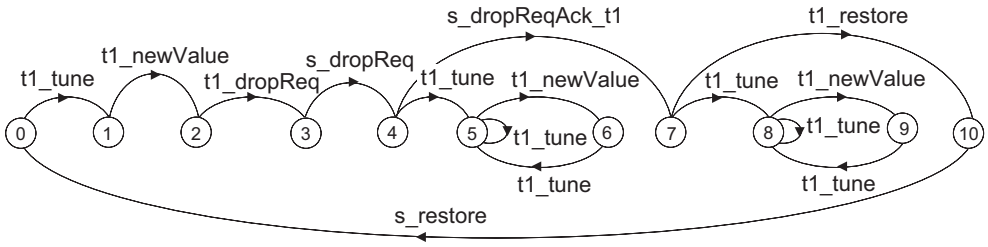


Figure 27: Pessimistic implementation of *It4*

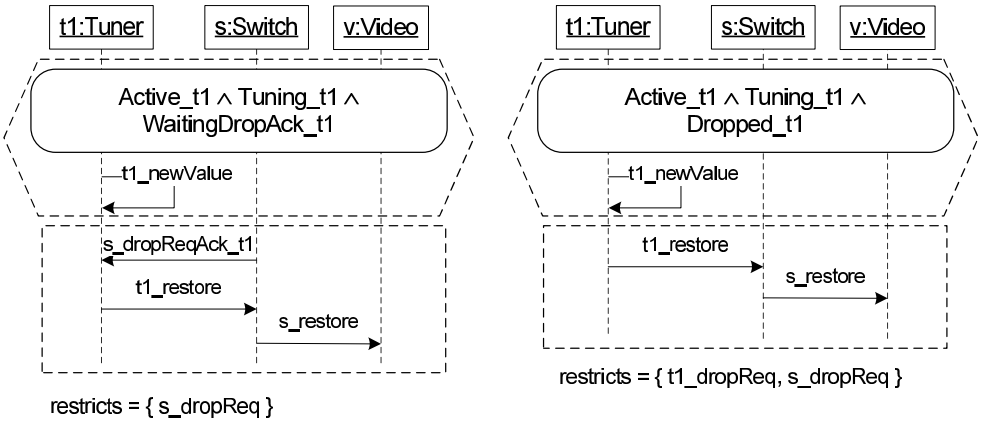


Figure 28: Scenarios enforcing the end of the protocol during a nested tune

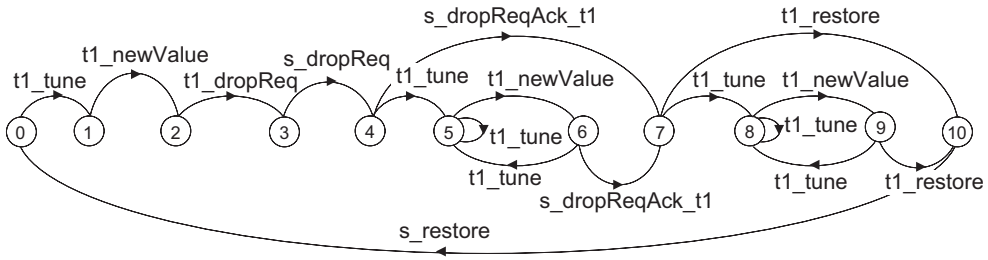


Figure 29: Pessimistic implementation of the resulting MTS after the final iteration

a maybe transition for requesting the signal be dropped. Something similar happens if we replay a trace leading to state 9 onto the synthesised MTS instead of its pessimistic version: there is a maybe transition restoring the signal. Hence, the scenario specification elaborated up to now is too weak and needs to be further elaborated so to make that *s_dropReqAck_t1* and *t1_restore* transition required when a nested tune occurs.

A further elaboration of the behaviour model for the protocol includes two eTS (Figure 28) to eliminate the problems observed in the previous iteration. The fluents *WaitingDropAck_t1* and *Dropped_t1* are used to identify each of the two store only sections, during the tuning protocol. The main charts in these recently defined scenarios show how the protocol is completed depending on which of the two store only sections the system is in.

The final iteration produces a model *It5* resulting from the merge of *It4* with the synthesised models from the two eTS in Figure 28. This model has only a few maybe transitions which, after experimenting with the prototype, we concluded should be refined into proscribed behaviour. Hence, we finalized the behaviour model elaboration process by selecting the pessimistic implementation of *It5* which is depicted in Figure 29. Validation of this model against the prototype did not prompt further changes.

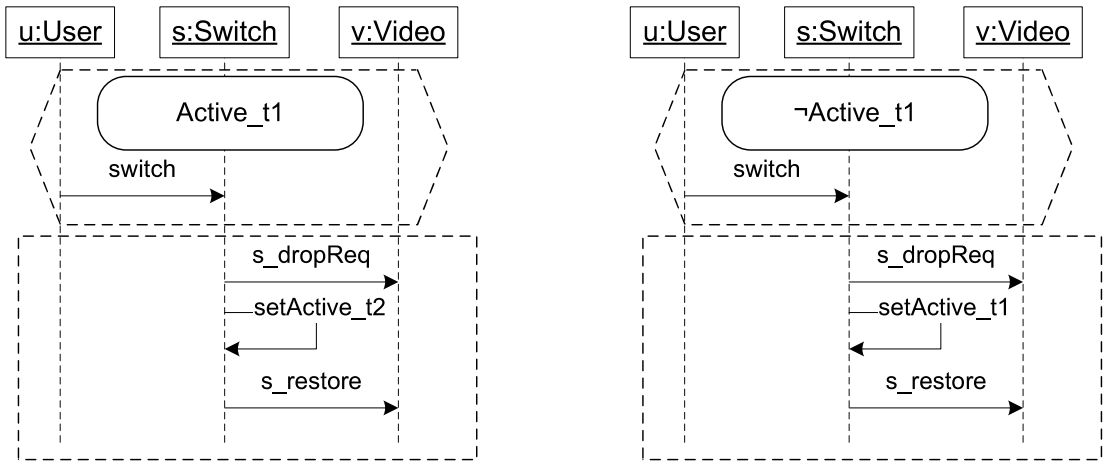


Figure 30: E_SwitchActive_t1 on the left and E_SwitchInactive_t1 on the right

5.4 Switching

Following a similar procedure as in the analysis of the tuning protocol, the prototype was used to analyse the behaviour of the protocol when switching tuners.

Initially, *t1* is the active tuner, and the occurrence of a *switch* triggers the following behaviour: A drop signal is sent to the video output, the signal of *t1* is replaced with that of *t2*, making tuner *t1* inactive and *t2* active, and finally the signal is restored to the video output. Once tuner *t2* is active, switching produces an analogous behaviour resulting in tuner *t1* as the active tuner and *t2* as the inactive one.

Two simple existential scenarios were created from these observations. One for the case when the tuner *t1* is active (*E_SwitchActive_t1*, on the left in Figure 30) and the other showing the case where *t2* is the active tuner (*E_SwitchInactive_t1*, on the right in Figure 30). The synthesised MTSs were then merged resulting in the partial model *It1* (Figure 31).

Note that state 5 of Figure 31 is where the trigger of *E_SwitchActive_t1* holds. The tuner *t1* is initially active and after a *switch* leading to state 5 the trigger holds. From that state there is a required path with the main chart of that scenario trough states 6 and 7 finishing at state 1. In this state, it is tuner *t2* that is active and taking a *switch* transition leads to state 2 whch triggers the main chart of scenario *E_SwitchInactive_t1*. The main chart of *E_SwitchInactive_t1* is satisfied by taking the required path through states 3 and 4 returning to the initial state.

Although states 6, 7, 3 and 4 exhibit required behaviour that reacts to *switch*, these states also have maybe *switch* transitions. These maybe transitions offer an opportunity for elaborating the behaviour of the description. Consider that, for example, from the initial state, where *t1* is the active tuner, and after a *switch* leading to state 5 it is possible to perform another *switch* and remain in state 5 from where there are required transitions *dropReq* and then *setActive_t2*. This means that switching twice does not lead to switching from *t1* to *t2* and back again to *t1*, but that the second switch is ignored, leading to *t2* being the active tuner after the two switches.

The situation described above could correspond to a requirement stating that if a user requests

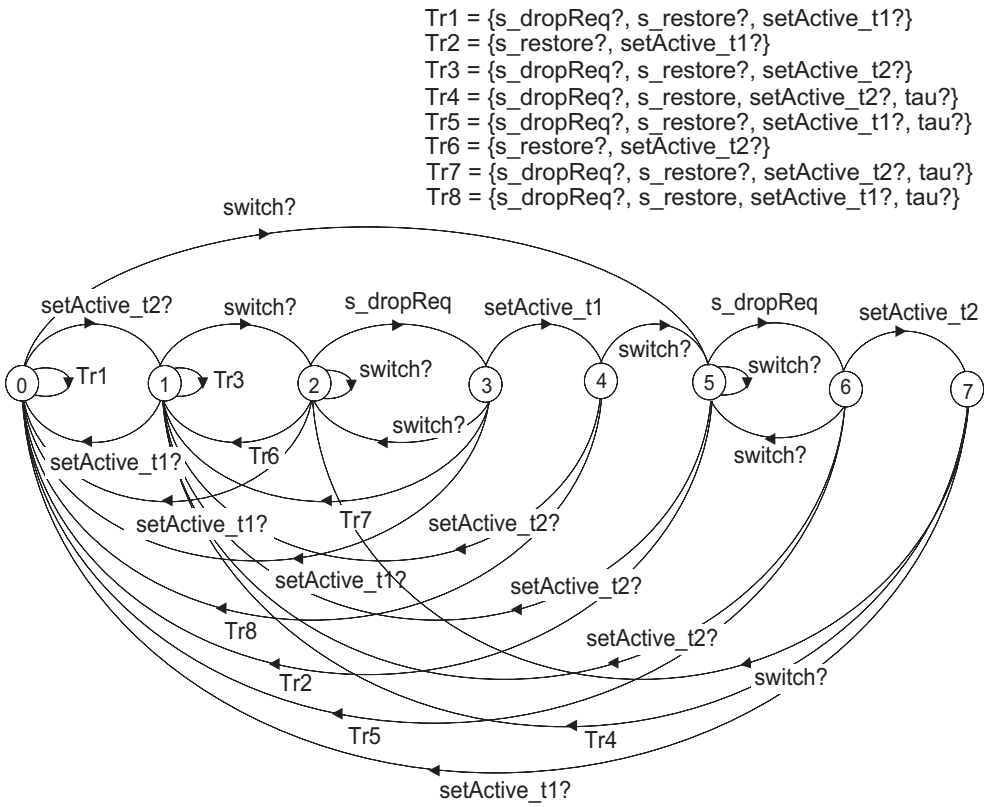


Figure 31: Resulting model from the first iteration: *It1*

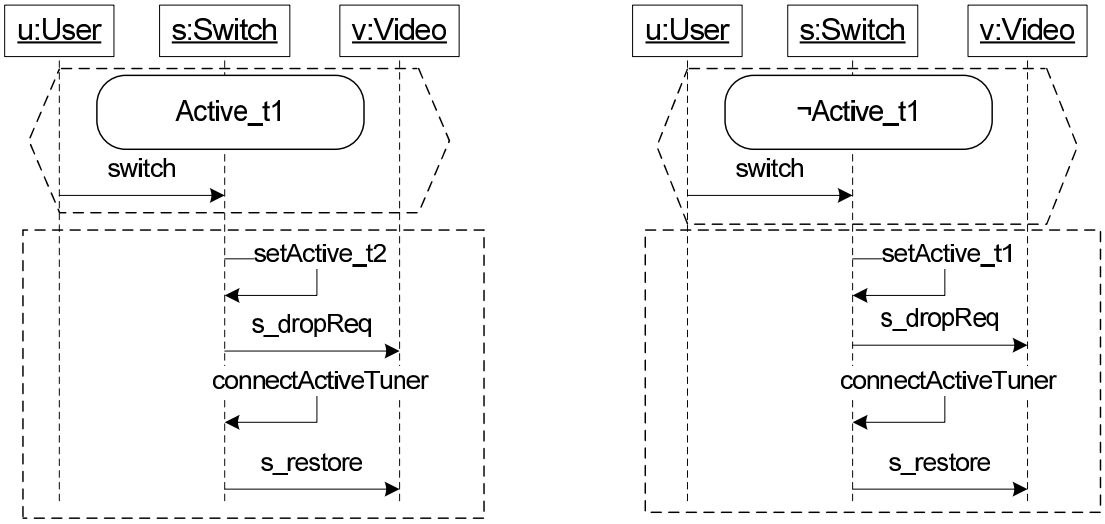


Figure 32: Modified version of E_SwitchActive_t1 on the left and E_SwitchInactive_t1 on the right

switching tuners during the processing of a previous switch request, the new switch request shall be ignored. Or, the scenario could simply indicate that the eTS produced does yet not adequately capture the intended system behaviour. As before, the prototype was used to provide domain knowledge.

In the prototype a nested switch is always allowed during this protocol. Moreover a switch *always* keeps track of the change of tuner and changes the signal only if needed. If the switch is performed several times before changing the signal then the signal is assigned to the active tuner. Therefore multiple switching has the same effect as performing the switches serially. The situations discussed above were therefore not intended system behaviour. The modified scenarios are shown in Figure 32. Unlike in previous scenarios, *setActive* now denotes activation of the tuner, and after the signal has been dropped the currently active tuner is connected (*connectActiveTuner*).

Note that the scenarios of Figure 32 must be existential because a new switch request should be allowed at any point of each main chart after the *setActive* action. In addition we want to reflect the alternating change of tuners and avoid traces like the one starting at state 0 with *set_Active_t1*? *set_Active_t1*?. Two universal scenarios are added (see Figure 33) to specify the alternating change of tuners.

Finally, to refine the maybe switch transitions of the MTS into required transitions based on the behaviour exhibited by the prototype, we used a fluent *Switching* to model the section of the protocol starting with a switch and ending with the occurrence of *set_Active_t1* or *set_Active_t2* and included an existential scenario *E_SwitchAllowed* (Figure 34) triggered by the condition ¬*Switching*. Partial models were then synthesised from the scenarios and merged leading to the MTS *It2* (Figure 35).

Analysis of the second iteration model of the protocol was performed through animation and

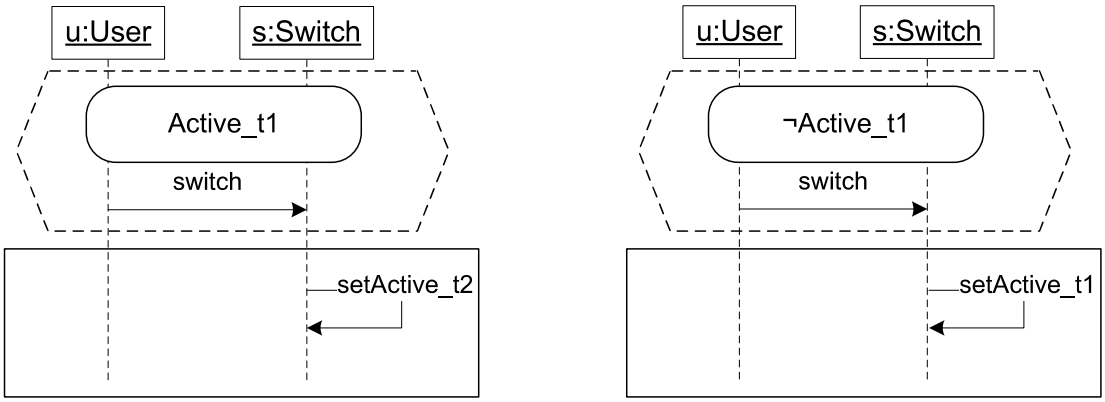


Figure 33: Universal scenarios for the alternating activation of the tuners

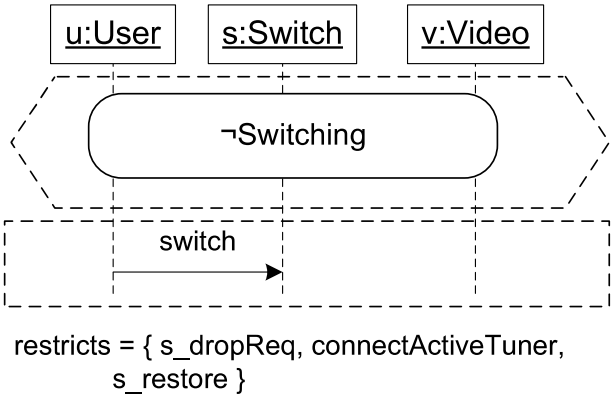


Figure 34: E_SwitchAllowed

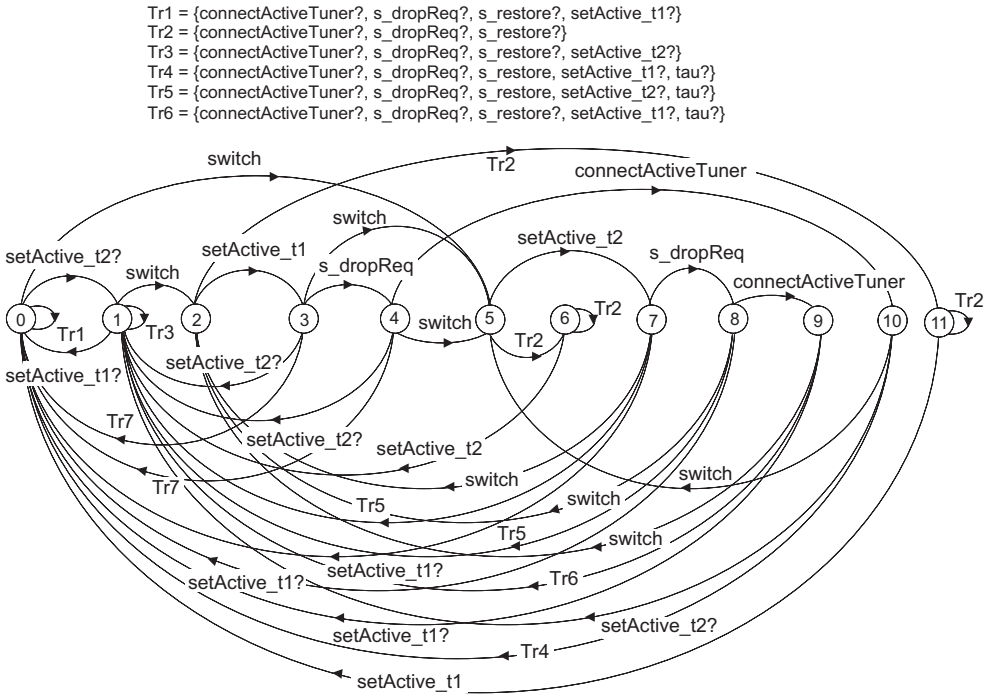


Figure 35: Resulting model from the second iteration: *It2*

resulted in discovering a required trace *switch setActive_t2 s_dropReq switch setActive_t1* starting at state 0 and leading to state 3. This describes how a second switch after the output signal has been dropped restarts the protocol. This situation can be better appreciated by analysing the pessimistic implementation of this partial model (Figure 36). There we can see that dropping the signal is required even when the signal has already been dropped. The prototype was used to validate if this was the intended system behaviour, resulting in the observation that a nested switch restarts the whole protocol only if the video output has not been dropped. Otherwise, if the signal is not being displayed in the video then the switch does not try to drop the signal and instead continues with the remaining section of the protocol.

In order to model this a fluent *SignalDropped* was created. The fluent is initially false as the video is displaying the frequency specified by the active tuner (*t1*). It becomes *true* with *s_dropReq* and *false* with *s_restore* when the signal is re-established. The scenarios *E_SwitchInactive_t1* and *E_SwitchActive_t1* are modified, strengthening their triggers to require that the signal is not dropped (Figure 37). To complete the specification two similar scenarios were added for the case when a switch occurs while the signal is dropped (Figure 38). In that case the protocol is exactly the same but the signal is not dropped. The scenarios are synthesised and merged leading to a model named *It3* (not shown).

Analysis of *It3* and validation of its few remaining maybe transitions against the prototype led to the conclusion that the pessimistic implementation of *It3*, shown in Figure 39, was an adequate model of the prototype. Validation of Figure 39 against the prototype did not prompt

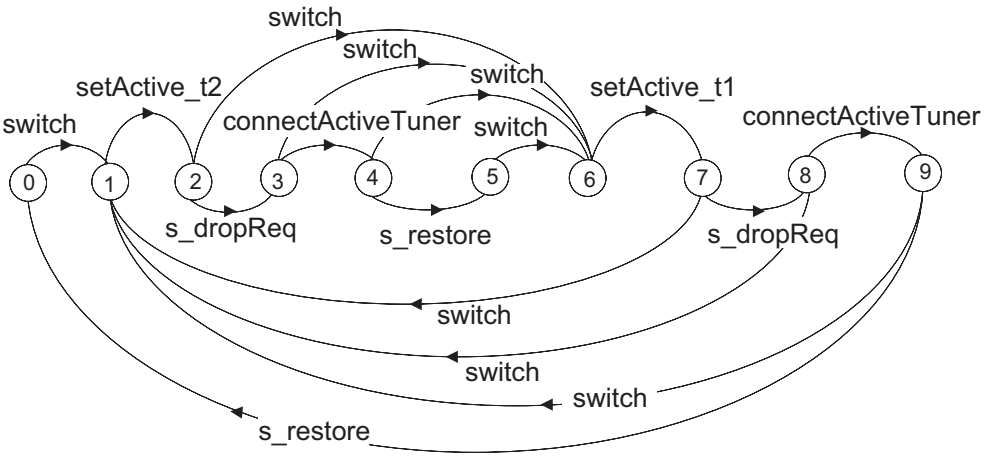


Figure 36: Pessimistic implementation of *It2*

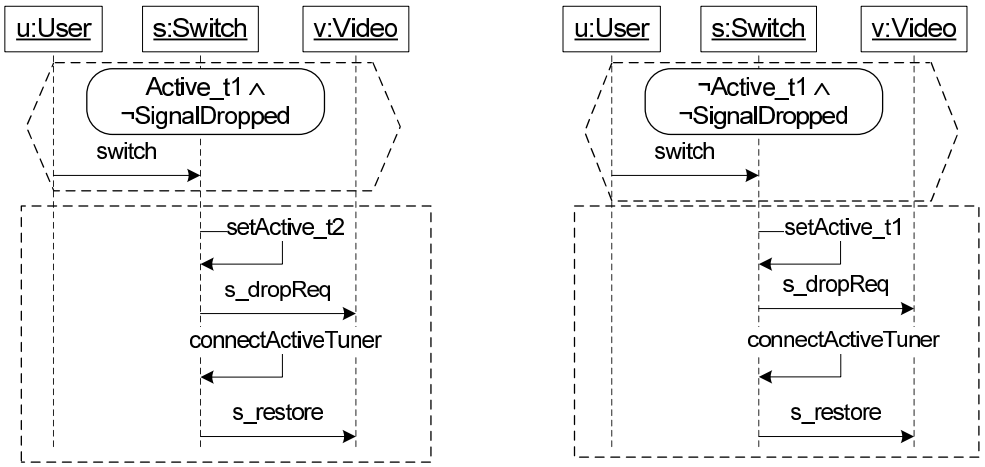


Figure 37: Final version of E_SwitchActive_t1 on the left and E_SwitchInactive_t1 on the right

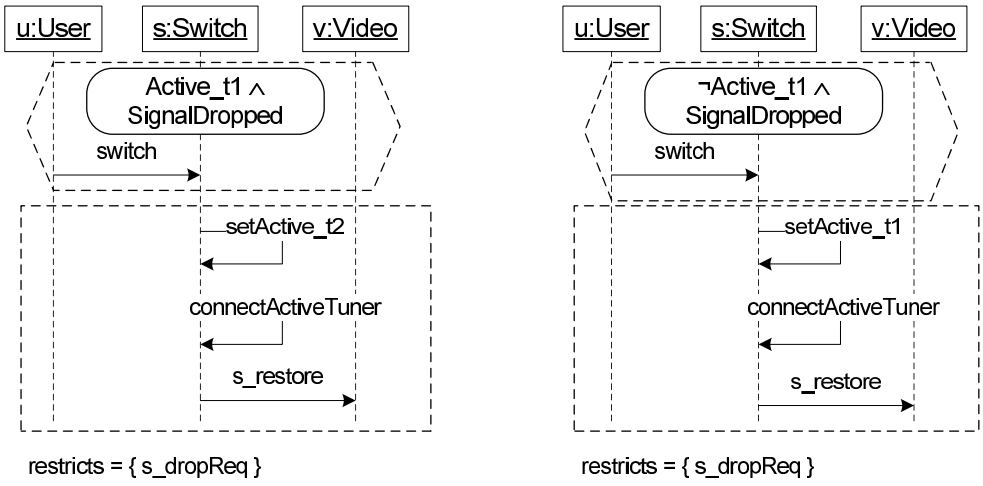


Figure 38: Scenarios showing how the switch works when the signal has been dropped

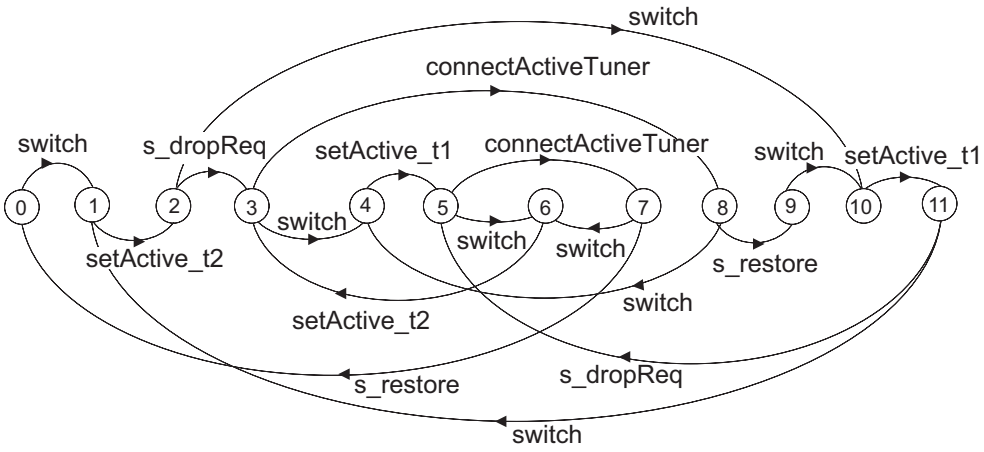


Figure 39: Pessimistic implementation of *It3*

further changes.

5.5 Case Study Conclusions

In this section, we reported on our experience using triggered scenarios and MTS synthesis to support the incremental elaboration of behaviour models. We described just one of the many elaborations which could have been performed as the result of analysing and eliciting behaviour from partial behaviour models. The stories we presented were somewhat simplified. In reality, we made numerous incorrect decisions in our understanding of the domain and in portraying this understanding in our scenarios and properties. We have reported on some aspects encountered to show how our approach supports exploring and validating behaviour.

The use of existential triggered scenarios was important for two reasons. Firstly, aspects of the behaviour of the subjects studied require triggers with an existential interpretation as opposed to a universal interpretation. For instance, the reaction of the system to user inputs was typically described with an existential scenario modelling the typical system response for the case that no further user inputs are provided. Note that, a simple-minded universal scenario would have proscribed the possibility of a repeated user input, as in the switch and tune actions. In fact, to avoid overconstraining the model it would require either the use of disjunction of all possible interactions from scratch or very low-grained scenarios showing state-based step-by-step progress. In this respect, existential scenarios provided a balanced means to express generalised rules of behaviour where the main chart is not intended to prescribe all possible future behaviours -just how the system must be able to progress in at least one possible future chain of messages.

Secondly, we found existentially triggered scenarios useful when producing first approximations of long interactions or complex descriptions. This is in-line with Damm and Harel's [31] position regarding behaviour model elaboration in which existential example-based descriptions are elaborated into universal rules that govern system behaviour. We found it convenient to start the elaboration process with existential scenarios, to synthesise them into one MTS for analysis. Typically we found it difficult to formulate universal scenarios with the right triggering condition and which avoided overrestricting intended system behaviour. Use of universal triggered scenarios early on can lead to unexpected chaining of triggers and main charts introducing unintended required behaviour. Keeping the yet-to-be validated behaviour as maybe behaviour allows a more guided elaboration strategy that is well served through the use of existential triggered scenarios.

Once the desired behaviour is more fully understood, universal statements, through general properties or universal triggered scenarios can be added to achieve a more aggressive prune of the set of valid implementations.

In our studies, we were able to reason about the multiple implementations that satisfy a partial specification (in the form of triggered scenarios) as a result of synthesising a single operational model that characterises all labelled transition systems that satisfy the specification. More specifically, the distinction between required, possible and proscribed behaviour that is offered by MTS allowed us to focus on the underspecified behaviour (the possible but not required behaviour), guiding the analysis and prompting questions aimed at completing the partial specification incrementally.

At each iteration, we were able to reason about the set of valid implementations using a variety of behaviour analysis techniques. In addition to model checking, we performed animations of the MTS models using the MTSA tool, exploiting their operational nature. We did not use graphical animation toolkits such as the one described in [32], because these have been designed for traditional behaviour models such as LTSs. However, we believe that these approaches can be adapted straightforwardly if some visual convention is used to distinguish between maybe and required behaviour. We also relied on inspection of synthesized MTSs, both in their textual and graphical forms, as produced by the MTSA tool. For larger models, validation of sliced, pessimistic and optimistic versions of the MTS were very helpful.

Note that forms of inspection of the MTS (or slices of it) support observation of the branching structure of the model; this is important in the context of a specification language that can express branching characteristics of system behaviour such as with eTS.

As mentioned previously, the analysis was to a large extent deliberately biased towards the maybe behaviour of the synthesised MTS. Producing traces that include maybe transitions helped in posing concrete questions for elicitation.

Triggers turned out to be one of the most interesting sources of analysis and elicitation. In fact, most of the manipulations done in the case studies could also be understood as detecting and solving issues linked to triggers that were either too weak or too strong.

It is worth pointing out that although the changes to the specification that were prompted by this elicitation (changes to existing scenarios or adding new ones) were local to a specific portion of the specification, the impact of the change in the resulting MTS was global. In other words, the further elaborated MTS is not the result of changing one or two maybe transitions to required transitions (or removing them all together). These global changes are a result of the various places at which triggers may be completed, forcing required behaviour, and more importantly, due to the chaining of triggers and main charts: a trigger that activates a main chart that, in turn, forces the occurrence of another trigger, etc.

Furthermore, the chaining of triggers led, in many cases, to the introduction of inconsistencies which were detected by MTSA as merge failure. Such inconsistencies led to the need to backtrack, removing scenarios one at a time, to explore the nature of the introduced inconsistency.

A more subtle situation that arose a number of times was that a triggered scenario was satisfied vacuously:where the only valid implementations are those that never trigger a specific scenario. We detected these by checking in the synthesised MTS that for every trigger in the specification, a trace exists that activates the trigger.

6 Discussion and Related Work

The approach presented above extends our previous work in [24] by providing universal triggered scenarios and an associated MTS synthesis algorithm, and by allowing the use of fluent expressions as conditions in both existential and universal triggered scenarios. The former is motivated by the need to provide a uniform framework that combines existential and universal scenarios to support moving from examples to comprehensive descriptions during the behaviour

elaboration process. The latter is motivated by our experience working on case studies which identified the need to have more expressive triggers to reduce the number of scenarios needed to describe the behaviour of a system-to-be.

A wide variety of scenario-based notations with diverse features and semantics have been developed. We focus our discussion on those with features that relate to triggers. 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 and thus are unable to mimic the eTS. Krüger [1] extends MSC with triggers and an associated universal semantics ("if a certain interaction pattern has occurred in the system, then another one is inevitable"). Sengupta and Cleaveland [7] also present a triggering mechanism with universal interpretation, but triggers are specified component-wise rather than system-wide. There is no support for existential scenarios with triggers. In the original formulation of LSCs [8], 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 construct. In fact, in later developments of LSCs (e.g. [4, 33]) the prechart for existential LSCs is dropped.

Although uTS defined in this paper are along similar lines to universal LSCs, the semantics is slightly different. Like a uLSC the main chart *must* follow the trigger. However, if the main chart's language has more than one linearisation then, in the case of uTS, *all* of the linearisations must be possible after the trigger. This is not the case of uLSC where the only condition is that after the prechart just one word in the main chart must follow. Consider the partially depicted computation tree of Figure 40. This tree violates the uTS in Figure 4 as once the trigger holds the interleaving in which *retainCard* holds before *alert* is not allowed. The set of words derived from the portion of the tree depicted does satisfy the same scenarios under uLSC semantics. Note that the semantics of uLSC and uTS is the same when the main chart's language is a singleton.

It is important to note that linear-time semantics of uLSCs cannot cannot be used as the semantics of uTS due to its branching nature. In addition, MTS are not sufficiently expressive to characterise the uLSC semantics (i.e. an MTS with exactly the same implementations as the set of LTS that satisfy the uLSC) as the latter requires at least one of the many linearisations. Such semantics could be captured, however, using Disjunctive MTS [34], a strictly more expressive variant of MTS. The synthesis algorithms presenting in this work would still be applicable in this context. DMTS may afford a number of advantages over MTS when used as the target formalism for synthesis. The study of DMTS in the context of synthesis is beyond the scope of this paper.

The semantics of eTS and uTS can be understood as a fragment of the temporal branching logic CTL. Informally, eTS stand for a formula of the form
$AG(\text{trigger holds} \rightarrow \bigwedge_{w \in L_M} EX\ \Phi_w)$ where $w = w_1 w_2 \dots w_k$, $\Phi_w = NU(w_1 \wedge (X(NU(w_2 \wedge (X(\dots)))))$ and $N = \bigwedge_{e \in \Sigma} \neg e$. Alternatively the semantics of uTS stand for a formula of the form
$AG(\text{trigger holds} \rightarrow (\bigwedge_{w \in L_M} EX\ \Phi_w) \wedge (AX \bigvee_{w \in L_M} \Phi_w))$. Once the trigger holds in a computation tree, in the case of eTS and uTS, at least one branch must exist for every word in the language of the main chart. The difference between eTS and uTS is that eTS allows branches

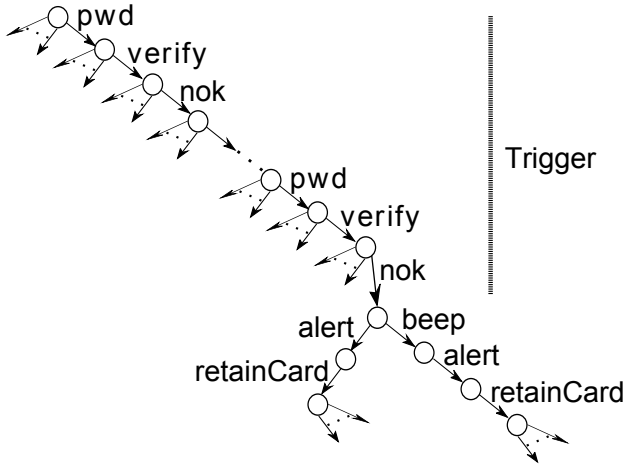


Figure 40: Part of an infinite computation tree satisfying uLSC interpretation of scenario in Figure 4 but violating its uTS interpretation

where the main chart does not follow and uTS forbids them. Summing up both of them require the possibility of branches with the interaction described in the main chart but in the case of uTS those are the only ones. Indeed, the semantics cannot be formulated in terms of the linear temporal logic LTL, traces or histories as can the semantics of uLSC [4] or the triggered MSC in [1]. TMSC in [7] also provides a branching flavoured semantics by using acceptance trees as semantic domain. There are several differences between TS and TMSC. TMSC without triggers are existentially interpreted, like MSC at early stages of system design [31].

They are combined in algebraic expressions describing the flow of control through a specification and in that sense they are similar to hMSC [6] (that flow is implicit in hMSC and explicit in TMSC expressions). The scenarios with triggers are used in TMSC expressions to eliminate nondeterminism which is the notion of refinement in that framework. So a TMSC with trigger is conjuncted to an expression leading to a new and more refined specification, i.e. with less nondeterminism. On the other hand each TS is a conditional rule over the whole system-to-be and the refinement notion is the refinement of MTS.

The notion of partial specification that we use is different from the one in [7]. In TMSC partial scenarios are described syntactically by not drawing a closing box at the end of an instance. The meaning is that the behaviour of that instance is unspecified after the TMSC ends and before the following TMSC starts so messages are allowed to be added by refining the scenario in a “fill in the blanks” fashion. In contrast TS are naturally partial as they have an associated alphabet and everything not in that alphabet can happen in between specified messages. Furthermore there is no restriction on the system’s behaviour after a main chart has been met.

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 little 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 [2, 5] provide synthesis algorithms of this characteristic.

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 [4], when restricted to uLSC, and [7] provide this style of synthesis.

In [17] 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 eTS and uTS (or eLSCs and uLSCs). In other words, it is not possible to build an LTS 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 [35] and [36].

In this paper, a three valued behaviour model is used as the target for synthesis. This step up in expressiveness allows the definition of a synthesis algorithm that characterises all LTS models that satisfy a TS.

This work is not the first to use partial behaviour models as the target for synthesis. The authors have previously studied synthesis of MTS from simple existential scenario descriptions (without triggers) and safety properties [17] exploiting the possibility of representing two bounds to system behaviour using MTS. These bounds are also exploited in [37] where MTSs synthesised from simple existentially interpreted sequence diagrams and a set of universally interpreted pre/post-conditions in the form of OCL constraints [38]. However, in [37] a distributed synthesis is performed: an MTS for each component present in the scenarios is synthesised and the parallel composition of these MTS is analysed for discrepancies between system-wide and component-level views (in a similar spirit to [2]). In this paper we propose a more expressive scenario language that could well be studied in a distributed synthesis setting such as [37].

Modal transition systems have been previously used as characterising sets of LTS but in a very different context. As noted in [39], one of the first attempts to apply modal transition systems was as the characterisation of the solutions of equation systems [34] involving bisimulation constraints with CCS-like context embedding an unknown process X. It turned out that a Disjunctive MTS characterises the set of all solutions to the equation system.

7 Conclusion

In this paper we have defined a scenario-specification language which includes support for describing triggered existential and universal scenarios. We have also defined a synthesis algorithm that constructs MTS models which characterise via refinement all LTS models that conform both to the existential and universal aspects of the scenario-based description.

A novel aspect of the approach is the use of triggered existential scenarios which have a branching semantics. This is in line with existing informal scenario-based and use-case based approaches to requirements engineering exploiting the expressive power of MTS in an operational behaviour model.

The approach supports behaviour elaboration through the analysis and refinement of underspecified system behaviour using MTS merging, model checking, inspection and animation, moving from examples to comprehensive descriptions during the behaviour elaboration process.

In future work, we intend to continue to develop and integrate support for elicitation and elaboration of behaviour models using MTS. In particular we are investigating the use of learning, in the form of Inductive Logic Programming [40], to aid the elaboration process. We aim to develop techniques and tools to support identifying, providing feedback and resolving inconsistencies in the process of merging MTS that result from scenario-based specifications.

Bibliography

[1] I. Kruger, "Distributed system design with message sequence charts," Ph.D. dissertation, Technical University of Munich, 2000.

[2] S. Uchitel, J. Kramer, and J. Magee, "Incremental elaboration of scenario-based specifications and behaviour models using implied scenarios," *ACM TOSEM*, vol. 13, no. 1, 2004.

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

[4] Y. Bontemps, P. Heymans, and P.-Y. Schobbens, "From live sequence charts to state machines and back: A guided tour," *IEEE Transactions on Software Engineering*, vol. 31, no. 12, pp. 999–1014, 2005.

[5] T. Ziadi, L. Helouet, and J.-M. Jezequel, "Revisiting statechart synthesis with an algebraic approach," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 242–251.

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

[7] B. Sengupta and R. Cleaveland, "Triggered message sequence charts," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 587–607, 2006.

[8] W. Damm and D. Harel, "LSCs: Breathing life into message sequence charts," in *FMOODS*, ser. IFIP Conference Proceedings, P. Ciancarini, A. Fantechi, and R. Gorrieri, Eds., vol. 139. Kluwer, 1999.

[9] K. Zachos, N. Maiden, and A. Tosar, "Rich-media scenarios for discovering requirements," *IEEE Software*, vol. 22, no. 5, pp. 89–97, 2005.

[10] R. M. Keller, "Formal verification of parallel programs," *Commun. ACM*, vol. 19, pp. 371–384, July 1976.

[11] R. Milner, *Communication and Concurrency*. New York: Prentice-Hall, 1989.

[12] K. Larsen and B. Thomsen, "A modal process logic," in *Proceedings of the Third Annual Symposium on Logic in Computer Science*, ser. LICS'88. IEEE Computer Society, 1988, pp. 203–210.

[13] K. Larsen, B. Steffen, and C. Weise, "The methodology of modal constraints," in *Formal Systems Specification*, ser. LNCS. Springer, 1996, vol. 1169, pp. 405–435.

[14] M. Huth, R. Jagadeesan, and D. A. Schmidt, "Modal transition systems: A foundation for three-valued program analysis," in *ESOP'01*, ser. LNCS, vol. 2028. Springer, 2001, pp. 155–169.

[15] K. G. Larsen, B. Steffen, and C. Weise, "A constraint oriented proof methodology based on modal transition systems," in *TACAS'95*, ser. LNCS. Springer-Verlag, 1995, pp. 13–28.

[16] D. Fischbein and S. Uchitel, "On correct and complete merging of partial behaviour models," in *Proceedings of SIGSOFT Conference on Foundations of Software Engineering*, ser. FSE'08. ACM, 2008, pp. 297–307.

[17] S. Uchitel, G. Brunet, and M. Chechik, "Synthesis of partial behaviour models from properties and scenarios," *IEEE Transactions on Software Engineering*, vol. 3, no. 35, pp. 384–406, 2009.

[18] G. Brunet, M. Chechik, D. Fischbein, N. D'Ippolito, and S. Uchitel, "Weak alphabet merging of partial behaviour models," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, to appear.

[19] G. Bruns and P. Godefroid, "Generalized model checking: Reasoning about partial state spaces," in *CONCUR'00*, ser. LNCS, vol. 1877. Springer-Verlag, 2000, pp. 168–182.

[20] D. Fischbein, N. D'Ippolito, G. Sibay, and S. Uchitel, "Modal Transition System Analyser (MTSA)," http://sourceforge.net/projects/mtsa/.

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

[22] S. Uchitel and M. Chechik, "Merging partial behavioural models," in *SIGSOFT '04/FSE-12*. ACM, 2004, pp. 43–52.

[23] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[24] G. Sibay, S. Uchitel, and V. Braberman, "Existential live sequence charts revisited," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 41–50.

[25] D. Giannakopoulou and J. Magee, "Fluent model checking for event-based systems," in *ESEC/FSE'03*. ACM, 2003.

[26] R. Alur, K. Etessami, and M. Yannakakis, "Inference of message sequence charts," *IEEE TSE*, vol. 29, pp. 623–633, July 2003.

[27] S. Mauw and M. A. Reniers, "High-level message sequence charts," in *International Conference on System Design Languages*, 1997, pp. 291–306.

[28] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh, "Efficient detection of vacuity in actl formulas," in *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV'97)*, ser. LNCS, vol. 1254. Springer-Verlag, 1997, pp. 279–290.

[29] N. D'Ippolito, D. Fishbein, H. Foster, and S. Uchitel, "MTSA: Eclipse support for modal transition systems construction, analysis and elaboration," in *eclipse '07: Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*. New York, NY, USA: ACM, 2007, pp. 6–10.

[30] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala component model for consumer electronics software," *Computer*, vol. 33, no. 3, pp. 78–85, 2000.

[31] W. Damm and D. Harel, "LSCs: Breathing life into message sequence charts," *FMSD*, vol. 19, no. 1, pp. 45–80, 2001.

[32] J. Magee, N. Pryce, D. Giannakopoulou, and J. Kramer, "Graphical animation of behavior models," in *ICSE'00*. ACM, 2000, pp. 499–508.

[33] H. Kugler, M. J. Stern, and E. J. A. Hubbard, "Testing scenario-based models," in *FASE*, ser. Lecture Notes in Computer Science, M. B. Dwyer and A. Lopes, Eds., vol. 4422. Springer, 2007, pp. 306–320.

[34] K. Larsen and L. Xinxin, "Equation solving using modal transition systems," in *LICS'90*. IEEE Computer Society, 1990, pp. 108–117.

[35] Y. Bontemps, P.-Y. Schobbens, and C. Löding, "Synthesis of open reactive systems from scenario-based specifications," *Fundam. Inform.*, vol. 62, no. 2, pp. 139–169, 2004.

[36] D. Harel and H. Kugler, "Synthesizing state-based object systems from lsc specifications," *Int. J. Found. Comput. Sci.*, vol. 13, no. 1, pp. 5–51, 2002.

[37] I. Krka, Y. Brun, G. Edwards, and N. Medvidovic, "Synthesizing partial component-level behavior models from system specifications," in *ESEC/FSE '09*. New York, NY, USA: ACM, 2009, pp. 305–314.

[38] D. Pilone and N. Pitman, *"UML 2.0 in a Nutshell"*. O'Reilly, 2005, http://www.uml.org/.

[39] A. Antonik, M. Huth, K. Larsen, U. Nyman, and A. Wasowski, "20 years of modal and mixed specifications," vol. 95, pp. 94–, Jun. 2008, columns: Concurrency.

[40] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel, "Learning operational requirements from goal models," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 265–275.

MEALS Partner Abbreviations

SAU: Saarland University, D

RWT: RWTH Aachen University, D

TUD: Technische Universität Dresden, D

INR: Institut National de Recherche en Informatique et en Automatique, FR

IMP: Imperial College of Science, Technology and Medicine, UK

ULEIC: University of Leicester, UK

TUE: Technische Universiteit Eindhoven, NL

UNC: Universidad Nacional de Córdoba, AR

UBA: Universidad de Buenos Aires, AR

UNR: Universidad Nacional de Río Cuarto, AR

ITBA: Instituto Técnológico Buenos Aires, AR