An Inductive Approach for Modal Transition System Refinement

Dalal Alrajeh[1], Jeff Kramer[1], Alessandra Russo[1], and Sebastian Uchitel[1,2]

1 Department of Computing, Imperial College London
180 Queens Gate, London SW7 2AZ, UK
{dalal.alrajeh04,j.kramer,a.ruso,s.uchitel}@imperial.ac.uk

2 Departamento de Computaciòn, FCEyN, UBA,
Buenos Aires, Argentina
suchitel@dc.uba.ar

Abstract

Modal Transition Systems (MTSs) provide an appropriate framework for modelling software behaviour when only a partial specification is available. A key characteristic of an MTS is that it explicitly models events that a system is required to provide and is proscribed from exhibiting, and those for which no specification is available, called *maybe* events. Incremental elaboration of maybe events into either required or proscribed events can be seen as a process of MTS refinement, resulting from extending a given partial specification with more information about the system behaviour. This paper focuses on providing automated support for computing strong refinements of an MTS with respect to event traces that describe required and proscribed behaviours using a non-monotonic inductive logic programming technique. A real case study is used to illustrate the practical application of the approach.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases Modal transition system, refinement, event calculus, inductive logic programming

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.106

1 Introduction

A Modal Transition System (MTS) is a state-transition formalism for specifying and verifying system behaviour. It extends conventional models such as Labelled Transition Systems (LTSs) by introducing modalities over transitions. Hence, an MTS not only models the events that a system is required to provide and is proscribed from exhibiting, but also explicitly models the events which system being modelled cannot guarantee to admit or prohibit, called *maybe* events.

Though MTSs have been introduced for over twenty years [9], it is only recently that the software engineering community has begun to develop automated support for stepwise elaboration of system requirements through MTSs. Intuitively, an MTS can be seen as a class of possible system implementations, each generated by changing maybe transitions into either required or proscribed. Within this context, a key notion is that of *modal refinement* [8]. Modal refinement is the process of incrementally refining an MTS, as more information about the system becomes available, by modifying possible behaviours into behaviours that *must* be provided or prevented by every system implementation of the given MTS. A final refined MTS would therefore be an LTS with just required events, where all unspecified events are assumed to be proscribed. We refer to this refined model as an *implementation*. Much work has been done on theoretical aspects of modal refinement and different notions of refinements have been presented (i.e. strong, weak and branching refinement[4]). However, how to compute these different refinements remains still an open problem.

The aim of our work is to provide a formal, tool-supported platform for incremental refinement of MTSs. This paper presents a first step towards such a general framework, in which it is shown how inductive learning can be used to compute *strong* refinements of MTSs from event traces. We consider a partial system description, consisting of a specific class of safety properties expressed in temporal logic, and assume that traces and the transitions to be required or proscribed are either provided by the user or automatically computed by model checking a given MTS with respect to some property. This description is encoded into an Event Calculus (EC) logic program, whose language is close to existing logic formalisms used in MTS synthesis and verification, and which allows explicit representation of event occurrences at different (time) points, through its time structure. We deploy a non-monotonic Inductive Logic Programming (ILP) system to generate new safety properties from event traces that would either require and proscribe transitions, and show how the learned properties characterise a *class* of implementations of the original MTS with respect to the given traces. The proposed approach is much influenced by our recent work on the elaboration of software requirements through ILP [1, 2] where we have shown how non-monotonic learning can be used to compute an implementation (i.e. an LTS) that satisfy a given set of properties from scenarios. In this paper, we show how inductive learning can be used to computes classes of implementations.

The paper is organised as follows. Section 2 describes background work on MTSs and the specification language used to construct and verify MTSs. Section 3 describes the proposed methodology. Section 4 presents a case study used to evaluate the our refinement process and Sections 5 and 6 conclude the paper with a discussion of related and future work.

2 Background

A Modal Transition System is a formalism used for modelling and reasoning about the behaviour of a system. A formal definition of an MTS is given below.

▶ **Definition 1** (Modal Transition System). A Modal Transition System is a tuple $M = (S, Act, \Delta^r, \Delta^p, s_0)$ where Act is the alphabet of label events, S a set of state, $\Delta^r \subseteq S \times Act \times S$ the set of *required* transitions, and $\Delta^p \subseteq S \times Act \times S$ the set of *possible* transitions, such that $\Delta^r \subseteq \Delta^p$. Transitions that are possible but not required are called *maybe* transitions. An MTS is said to have a required transition on a, denoted $s \xrightarrow{a}_r s'$, if $(s, a, s') \in \Delta^r$. It is said to have a possible transition on a, denoted $s \xrightarrow{a}_p s'$, if $(s, a, s') \in \Delta^p$.

An implementation is an MTS $(S, Act, \Delta^r, \Delta^p, s_0)$ where $\Delta^r = \Delta^p$, also referred to as an LTS. An MTS can be represented as a directed graph in which nodes correspond to states and the edges between two nodes represent the transition relation. Maybe transitions are denoted with a question mark following the event label. System executions are represented as sequences of transitions called *traces*. A trace can be either required, possible or proscribed. A formal definition is given below. The definition of a trace presupposes that only one event can occur at a single position in a trace, i.e. events cannot occur simultaneously.

▶ **Definition 2** (Traces). Let $M = (S, Act, \Delta^r, \Delta^p, s_0)$ be an MTS. A trace $\sigma = a_1, a_2, ...$ where $a_i \in Act$ is said to be a *required trace in* M if there exists in M an infinite sequence of transitions $s_0 \xrightarrow{a_1}_r s_1, s_1 \xrightarrow{a_2}_r s_2 ...$; it is said to be a *possible trace in* M if there exists in M

an infinite sequence of transitions $s_0 \xrightarrow{a_1}_p s_1, s_1 \xrightarrow{a_2}_p s_2 ...$, where $(s_i, a_{i+1}, s_{i+1}) \in \Delta^p$ for each $i \geq 0$, and there is at least one transition relation in the sequence that is in $\Delta^p - \Delta^r$. Otherwise, it is said to be a *proscribed trace in M.*

In this paper, we refer to traces that should be possible in a model as *positive* traces, and to traces that should be proscribed as *negative* traces. The notion of *modal refinement* is defined between two MTSs and states when one MTS is "more defined" than another. Given two MTSs N and M, N is said to *refine M* if N preserves all the required and all the proscribed transitions of M. In this paper we assume a particular notion of refinement relation, called *strong refinement*, which assumes that all MTSs share the same *alphabet* [9]. This is defined as follows.

▶ **Definition 3** (Strong Refinement). Let $\wp$ be the universe of all MTSs for a given alphabet *Act*. An MTS $N = (U, Act, \delta^r, \delta^p, u_0)$ is a refinement of an MTS $M = (S, Act, \Delta^r, \Delta^p, s_0)$, written as $M \preceq N$, if there exists some refinement relation $\mathcal{R} \subseteq S \times U$ such that, for all $s \in S$ and $u \in U$, if $(s, u) \in \mathcal{R}$, the following holds for every label a in *Act*:

- if $(s \xrightarrow{a}_r s')$, then for some $u' \in U$, $(u \xrightarrow{a}_r u' \ \wedge \ (s', u') \in \mathcal{R})$; and
- if $(u \xrightarrow{a}_p u')$, then for some $s' \in S$, $(s \xrightarrow{a}_p s' \ \wedge \ (s', u') \in \mathcal{R})$.

An MTS can be synthesised from and verified against formulae expressed in some form of temporal logic. We give here a brief description of FLTL[7] as the language used by the MTSA model checker [3]. In FLTL, a fluent f is defined by a tuple consisting of a set I_f of initiating events, a set T_f of terminating events and an initial truth value (**tt** or **ff**), such that $I_f \subseteq Act$, $T_f \subseteq Act$ and $I_f \cap T_f = \emptyset$. We write $f = \langle I_f, T_f, Init \rangle$ as a shorthand for a fluent definition, where $Init \in \{\mathbf{tt}, \mathbf{ff}\}$. Every event label $a \in Act$ defines a fluent $\dot{a} = \langle a, Act \backslash \{a\}, \mathbf{ff} \rangle$. We refer to such fluents as event fluents.

Given the set of fluents F, FLTL formulae can be constructed using the standard boolean connectives and temporal operators

$$\mathbf{tt} \mid \mathbf{ff} \mid f \mid \neg\phi \mid \phi \vee \psi \mid \phi \wedge \psi \mid \mathsf{X}\phi \mid \phi \mathsf{U} \psi \mid \mathsf{F}\phi \mid \mathsf{G}\phi$$

Given a set of traces Σ over *Act* and a set D of fluent definitions, a fluent is said to be true in a given trace $\sigma = \langle a_1, ..., a_n \rangle$ at position i with respect to D, denoted $\sigma, i \models_D f$, if and only if either of the following conditions hold:

- f is defined initially true and $\forall j \in \mathcal{N}. \ ((0 < j \leq i) \rightarrow a_j \notin T_f)$;
- $(\exists j \in \mathcal{N}. \ (j \leq i) \wedge (a_j \in I_f)) \wedge (\forall k \in \mathcal{N}.((j < k \leq i) \rightarrow a_k \notin T_f))$.

In other words, a fluent f holds if and only if it is initially true or an initiating event for f has occurred, and no terminating event has occurred since. The semantics of boolean connectives are defined in the standard way. The semantics of the temporal operators are defined as follows:

$\sigma, i \models_D \mathsf{X}\phi$, *if and only if* $\sigma, i+1 \models_D \phi$
$\sigma, i \models_D \phi \mathsf{U} \psi$, *if and only if* $\exists j \geq i. \ \sigma, j \models_D \psi$ *and* $\forall i \leq k < j. \ \sigma, k \models_D \phi$
$\sigma, i \models_D \mathsf{G}\phi$, *if and only if* $\forall j \geq i. \ \sigma, j \models_D \phi$
$\sigma, i \models_D \mathsf{F}\phi$, *if and only if* $\exists j \geq i. \ \sigma, j \models_D \phi$

Given that MTSs represent a set of possible implementations, it is often necessary to reason about properties that hold in some or all implementations or none. Thus the satisfaction of FLTL formulae over MTSs is given a 3-valued semantics. An MTS M is said to satisfy a property ϕ with respect to D, if ϕ is satisfied in every possible trace of M with respect to D. It is said to be violated in M, if there is a required trace in M that refutes it or if all possible traces in M violate it. Otherwise, the satisfaction of ϕ is *unknown*, meaning that some implementations satisfy ϕ while others do not. Two FLTL formulae ϕ and ψ are consistent if there exists a model M that satisfies the formula $\phi \wedge \psi$. For the remainder of this paper, we use $\wp(\phi)$ to denote an MTS that satisfies ϕ with respect to D. Furthermore, we consider only safety properties of the form $(\mathsf{G}\ (\bigwedge_{0\leq i} f_i \rightarrow \mathsf{X}(\neg)\dot{a}))$ where f_i is a literal over event or non-event fluent, and a is an event label. We also assume that the MTS synthesised from given safety properties is the least refined MTS that satisfies the given properties (for further detail see [13]).

3 Approach

The aim of our work is to develop an automated approach for refining MTSs, given a set of traces that represent required and proscribed system behaviour. In this paper we focus on the notion of strong refinement given in Definition 3 and on FLTL safety properties of the form $(\mathsf{G}\ (\bigwedge_{0\leq i} f_i \rightarrow \mathsf{X}(\neg)\dot{a}))$ described above. The input to our approach is a set D of fluent definitions, a set $\Gamma = \{\gamma_i\}$ of safety properties and two disjoint sets of positive and negative traces, $\Sigma = \Sigma^+ \cup \Sigma^-$ that are possible traces in the MTS M synthesised from Γ. The input D, Γ and Σ are encoded into an Event Calculus (EC) logic program, and a non-monotonic ILP system is used to learn rules about required and proscribe transitions that can be translated back into FLTL safety properties Φ satisfying the following property: the MTS N synthesised from the (refined) property $\bigwedge \gamma_i \wedge \Phi$ is a strong refinement of the given MTS M where every trace in Σ^+ is a possible trace in N and every trace in Σ^- is a proscribed trace in N.

▶ **Definition 4** (Refinement with respect to traces)**.** Let $M = \langle S, Act, \Delta^r, \Delta^p, s_0\rangle$ and let $\Sigma = \Sigma^+ \cup \Sigma^-$ be a set of positive and negative traces that are possible in M. An MTS $N = \langle U, Act, \delta^r, \delta^p, u_0\rangle$ is a *correct refinement of M with respect to* Σ if and only if N is a refinement of M ($M \preceq N$) and every trace $\sigma^+ \in \Sigma^+$ is a possible trace in N, and every trace $\sigma^- \in \Sigma^-$ is a proscribed trace in N.

▶ **Definition 5** (Refinement Task)**.** Let $\Gamma = \{\gamma_1, ..., \gamma_n\}$ be a set of FLTL safety properties, let D a set of fluent definitions, let $M = \wp(\bigwedge \gamma_i)$ be an MTS that satisfies Γ w.r.t D and $\Sigma = \Sigma^+ \cup \Sigma^-$ a set of positive and negative traces that are possible in M. The refinement task is to find an FLTL safety property Φ such that the MTS $N = \wp(\bigwedge \gamma_i \wedge \Phi)$ is a correct refinement of M with respect to the traces in Σ. We call Φ a *consistent extension* of Γ.

3.1 An Event Calculus for MTSs

The Event Calculus is a widely-used logic programming formalism for reasoning about actions and time [12]. The definition of an EC language in this paper includes terms of four different types: *event* terms, *fluent* terms, *time* (here referred as position) terms and *trace* terms. Position terms are represented by the non-negative integers $0, 1, 2, \ldots,$ events correspond to actions that can be performed, fluents correspond to time-varying Boolean properties, and traces are constants denoting different (independent) time lines.

The EC ontology includes the basic predicates *happens*, *initiates*, *terminates*, *initially* and *holdsAt*. The atom *happens*(e, p, c) indicates that event e occurs at position p in a trace c, *initiates*(e, f) (resp. *terminates*(e, f)) means that, if event e were to occur, it would cause fluent f to be true (resp. false) immediately afterwards. The predicate *holdsAt*(f, p, c)

indicates that fluent f is true at position p in a trace c, and *initially*(f) means that fluent f is initially true. The formalism also includes an auxiliary predicate *clipped*(p_1, f, p_2, c) which is defined as an event that terminates f occurs between positions p_1 and p_2 in a trace c. The interactions between these EC predicates are governed by the set of domain-independent core axioms defined below, where *not* denotes negation by failure.

```
clipped(P1,F,P2,C):- happens(E, P, C), terminates(E, F), P1 ≤ P<P2.
   holdsAt(F,P2,C):- happens(E,P1,C), initiates(E,F),                  (1)
                     P1 < P2, not clipped(P1,F,P2,C).
   holdsAt(F,P,C) :- initially(F), not clipped(0,F,P,C).
```

To capture in EC the different types of MTS transitions we have extended the language with the predicates *required*, *proscribed*, and *maybe*. The atom *required*(e, p, c) (resp. *proscribed*(e, p, c)) means that event e's occurrence is required (resp. proscribed) at position p in a given trace c. The atom *maybe*(e, p, c) is defined in (2) below and means that the occurrence of event e at position p in trace c has not yet been specified.

```
     maybe(E,P,C):- not required(E,P,C), not proscribed(E,P,C).        (2)
```

Auxiliary predicates are also introduced to refine and appropriately constraint the notion of occurrence of event: *req_happens*(e, p, c) is used to capture the fact that all required events must happen, and *may_happens*(e, p, c) defines that a maybe event may happen at some position in a trace if executed at that position in that trace. They are related to the *happen* predicate by the following axioms:

```
 req_happens(E,P,C):- required(E,P,C).                                 (3)
may_happens(E,P,C):-  executed(E,P,C), maybe(E,P,C).                   (4)
    happens(E,P,C):-  may_happens(E,P,C).                              (5)
     happens(E,P,C):- req_happens(E,P,C).                              (6)
```

Integrity constraints over these predicates are captured by the following denial rules[1]:

```
false:- required(E,P,C), proscribed(E,P,C).                            (7)
false:- required(E,P,C), not executed(E,P,C).                          (8)
false:- happens(E1,P,C), happens(E2,P,C), not eq(E1,E2).               (9)
```

Constraint (7) states that an event cannot be required and proscribed at the same position in a trace, whereas constraint(8) states that a required event must be executed at the position where it is required. A weaker semantics for required transitions is discussed in Section 5.

In addition to the above domain-independent axioms, our EC program is equipped with *domain-dependent axioms* that capture properties of the given partial system description. The following definition shows how FLTL partial specifications are encoded into domain-dependent EC axioms.

▶ **Definition 6** (EC Encoding). Let $\Gamma = \{\gamma_1, ..., \gamma_n\}$ be a set of safety properties, D a set of fluent definitions and Σ a set of finite traces, such that every trace in Σ is a possible trace in MTS $M = \wp(\bigwedge \gamma_i)$. The EC encoding of Γ and D, denoted $EC(\Gamma \cup D \cup \Sigma)$, is the EC program Π constructed as follows:

[1] Denial rules take the form `false :- (not)b_1, ..., (not)b_n` where *b_i* can be any atom defined in the language.

- add to Π, for each fluent definition $f = \langle\{a_i\}, \{b_i\}, Init\rangle$ in D,
  - the set of facts $initiates(a_i, f)$ and $terminates(b_i, f)$,
  - the fact $initially(f)$ if f is defined in D as initially true,
- add to Π, for each safety property of the form $\Box(\bigwedge_{0\leq i\leq k}(\neg)f_i \rightarrow \bigcirc\neg\dot{a})$, the rule $proscribed(a, P, C)$:-$(not)\ holdsAt(f_1, P, C), ..., (not)\ holdsAt(f_k, P, C)$[2],
- add to Π, for each safety property of the form $\Box(\bigwedge_{0\leq i\leq k}(\neg)f_i \rightarrow \bigcirc\dot{a})$, the rule $required(a, P, C)$:-$(not)\ holdsAt(f_1, P, C), ..., (not)\ holdsAt(f_k, P, C)$,
- add to Π, for each trace $\sigma_j = \langle a_1, ..., a_n\rangle \in \Sigma$, the set of facts $executed(a_i, i-1, c_j)$, where $0 < i \leq n$ and c_j denotes the trace σ_j.

The EC program described above is a *normal logic program* for which we use the notions of stable model semantics and entailment under credulous stable model semantics [6]. Hence we say that an EC program Π *entails* an expression π, denoted $\Pi \models \pi$, if and only if π is satisfied in at least one stable model of Π. The following theorem proves that the above EC encoding is sound.

▶ **Theorem 7** (Soundness of EC Encoding). *Let Γ be a set of safety properties, D be a set of fluent definitions and $M = (S, Act, \Delta^r, \Delta^p, s_0)$ an MTS that satisfies Γ with respect to D. Let $\sigma = \langle a_1, ..., a_n\rangle$ be a finite possible trace in M. Let Π be the EC logic program given by $EC(\Gamma \cup D \cup \sigma)$ and let I be a stable model of Π. Then, for each fluent f in D and position p in σ, where $0 \leq p \leq n$, f is true at position p in σ if and only if $holdsAt(f, p, \sigma) \in I$; for every event $a \in Act$ and position p in the trace σ, where $0 \leq p \leq n$, there is a required transition on a at position p in M if and only if $req\_happens(a, p-1, \sigma) \in I$ and there is a maybe transition on a at p in M if and only if $may\_happens(a, p-1, \sigma) \in I$.*

The proof is by induction of the position p in the trace σ, using the fact that Π is a locally stratified program and, as such, has a unique stable model.

3.2 Refining MTS using Inductive Logic Programming

Inductive Logic Programming (ILP) is concerned with the computation of hypotheses H that extend a prior background theory B to entail a set of examples E, i.e. $B \cup H \models E$ [10]. The hypotheses H are assumed to be part of a set of clauses HS, called the *hypothesis space*, which defines all hypotheses that would be accepted as a solution.

▶ **Definition 8** (Inductive Solution). Given a normal logic program B, a set of ground literals E, and a set of clauses HS, the task of ILP is to find a normal logic program $H \subseteq HS$, consistent with B, such that $B \cup H \models E$ under the stable model semantics. H is called an *inductive solution* for E with respect to B and HS.

In our refinement approach, the inductive learning task is to compute hypotheses H from a background B, given by the EC encoding of a partial system specifications, that is consistent with the given specification, and that entails, together with B, require and proscribe event transitions specified in a given set of traces. The (possible) traces in a given Σ are therefore translated into facts about what is required to happen and what should be proscribed from happening in the refined model. These facts constitute the set E of examples for our inductive learning task.

[2] (not) preceding a literal is a shorthand for either the positive or negative form of that literal.

▶ **Definition 9** (EC encoding of event traces into examples). Let $\Gamma = \{\gamma_1, ..., \gamma_n\}$ be a set of safety properties, D a set of fluent definitions and $\Sigma^+ \cup \Sigma^-$ a set of finite positive and negative traces, such that every trace in Σ is a possible trace in MTS $M = \wp(\bigwedge \gamma_i)$. The EC translation of traces in Σ into examples E, denoted EC(Σ), is constructed as follows:

- for each trace $\sigma_j^+ = \langle a_1, ..., a_m \rangle \in \Sigma^+$ where $s_0 \xrightarrow{a_1}_p s_1, ..., s_{m-1} \xrightarrow{a_m}_p s_m$ in M
  - add to E facts *req_happens*$(a_i, i-1, c_j)$ for every transition $s_{i-1} \xrightarrow{a_i}_p s_i$ that should be required, where $1 \leq i \leq m$,
  - add to E facts *happens*$(a_i, i-1, c_j)$ for all other transitions, where $1 \leq i < m$,
- for each trace $\sigma_k^- = \langle b_1, ..., b_n \rangle \in \Sigma^-$, where $s_0 \xrightarrow{b_1}_p s_1, ..., s_{n-1} \xrightarrow{b_n}_p s_n$ in M
  - add to E the fact *not happens*$(b_n, n-1, c_k)$.

So transitions in the traces of Σ^+ that are intended to be required in the refined model are formalised as *req_happens* atoms, and transitions in the traces of Σ^- that are intended to be proscribed are encoded with *not happens* literals. All other transitions that are not defined as *required* or *proscribed*, are represented as *happens* atoms.

To learn safety properties rules, we defined the HS to be the set of clauses with *proscribed* or *required* in the head and *holdsAt* literals in the body. Because the examples are given in terms of *req_happens* and (*not*) *happens*, and given our EC background knowledge, our learning task requires an ILP algorithm capable of non-observational predicate and non-monotonic learning. For this we have used the ILP system in [11].

In brief, for every transition that is assumed to be executed, and for which the background knowldge B does not include any proscribed rule, the consequence that it may happen can be derived from the background knowledge. This is inconsistent with the examples *req_happens* or *not happens* given in E for the same event. So the learning algorithm explains the example (consistently) by abducing a set A of ground required and proscribed facts for these transitions. These then are used to form the head of the learned rules. Body literals are grounds *holdsAt* literals derived from $B \cup A$ at the same positions of the traces of the adbuced literals. These constructed ground rules are then generalised in a way that preserves consistency with the integrity constraints included in the background knowledge. The learning may produce alternative solutions. Once a solution H is chosen, this is translated back into FLTL. Rules of the form *proscribed*(a, P, C):-(*not*) *holdsAt*$(f_1, P, C), ...,$ (*not*) *holdsAt*(f_k, P, C) are translated back into safety properties of the form $\Box(\bigwedge_{0 \leq i \leq k}(\neg)f_i \rightarrow \bigcirc\neg\dot{a})$, while rules of the form *required*(a, P, C):-(*not*) *holdsAt*$(f_1, P, C), ...,$ (*not*) *holdsAt*(f_k, P, C) are translated back into FLTL properties of the form $\Box(\bigwedge_{0 \leq i \leq k}(\neg)f_i \rightarrow \bigcirc\dot{a})$. The soundness of the learning step is proved by Lemma 10 and Theorem 11.

▶ **Lemma 10.** *Let $\Gamma = \{\gamma_1, ..., \gamma_n\}$ be a set of properties expressed in FLTL, D a set of fluent definitions and $\Sigma = \Sigma^+ \cup \Sigma^-$ a set of traces, such that every trace in Σ is a possible trace in the MTS $M = \wp(\bigwedge \gamma_i)$. Let $\Pi = EC(\Gamma \cup D \cup \Sigma)$ be the EC translation of Σ, D and Γ, and let $E = EC(\Sigma)$ be the encoding of Σ into examples. Let H be an inductive solution for E with respect to Π, within the hypothesis space HS. Then FLTL(H) is consistent with the given specification Γ.*

The proof is by contradiction were Γ and FLTL(H) are assumed to be inconsistent and show that it falsifies the assumption of H be an inductive solution.

▶ **Theorem 11.** *Let $\Gamma = \{\gamma_1, ..., \gamma_n\}$ be a set of safety properties expressed in FLTL, D a set of fluent definitions and $\Sigma = \Sigma^+ \cup \Sigma^-$ a set of traces, such that every trace in Σ is a possible trace in the MTS $M = \wp(\bigwedge \gamma_i)$. Let $\Pi = EC(\Gamma \cup D \cup \Sigma)$ be the EC translation of Σ, D and Γ, and let $E = EC(\Sigma)$ be the encoding of Σ into examples. Let H be an inductive solution* *for E with respect to Π, under the hypothesis space HS. Then FLTL(H) is a consistent extension of Γ with respect to the traces in Σ.*

4 A Case Study

In this section we illustrate an application of our approach to a real case study, the Philips Television Set Configuration reported in [14], which describes an industrial protocol for a product family of Philips television sets. We used the MTSA tool described in [3] to construct an MTS from the available (partial) system description and to verify the resulting refined MTS.

The system comprises multiple tuners, a video output device and a switch that can be configured by the television user to display several signals in different configurations. The protocol is concerned with controlling the signal path to avoid visual artefacts appearing on video outputs when a tuner is changing frequency. The alphabet is composed of $\{set\_Active\_t1, set\_Active\_t2, t1\_tune, s\_restore, t1\_dropReq, s\_dropAck\_t1\}$. The fluent definitions are given by the following tuples:

Active_t1 =<*set_Active_t1, set_Active_t2, tt*>
Tuning_t1 =<*t1_tune, {s_restore, set_Active_t1, set_Active_t2}, ff*>
WaitingDropAck_t1 =<*t1_dropReq, s_dropAck_t1, ff*>
Dropped_t1=<*s_dropAck_t1, t1_restore, ff*>

Figure 1 shows an MTS generated from the initial descriptions. Note that the numbered nodes are used for reference and do not designate a particular state in Δ^p. The positive and negative traces $\Sigma^+ \cup \Sigma^-$ include:

$$\Sigma^+ = \{\langle start, t1\_tune\rangle\} \tag{10}$$

$$\Sigma^- = \{\langle start, t1\_tune, t1\_newValue, t1\_tune\rangle, \langle start, t1\_tune, t1\_newValue, t1\_dropReq, s\_dropAck\_t1, t1\_restore, t1\_tune\rangle\} \tag{11}$$

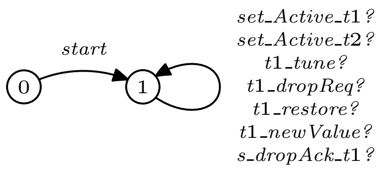

Figure 1 **An initial MTS M for the Philip TV set**

In the traces given in Σ the proscribed transitions are last occurrence of transition *t1_tune* in each trace in Σ^-, whereas the required transitions are the transition *t1_tune* in Σ^+. To learn safety properties that would ensure that only those MTS refinements that would meet these proscribed and required transitions are generated, we translate the given fluent definitions into EC domain-dependent axioms and the given traces in Σ into EC narratives and examples as defined in Section 3. Part of this translation is given below.

```
initiates(set_Active_t1,active_t1).     terminates(set_Active_t2,active_t1).
initially(active_t1).
initiates(t1_tune,tuning_t1).           terminates(s_restore,tuning_t1).
terminates(set_Active_t1,tuning_t1).    terminates(set_Active_t2,tuning_t1).
initiates(s_dropReqAck_t1,dropped_t1).     terminates(t1_restore,dropped_t1).
initiates(t1_dropReq,waitingDropAck_t1).terminates(s_dropAck_t1,waitingDropAck_t1).

executed(start,0,c1). executed(t1_tune,1,c1).
executed(start,0,c2). executed(t1_tune,1,c2).
executed(t1_newValue,2,c2).

    executed(t1_tune,3,c2). executed(start,0,c3).
    executed(t1_tune,1,c3). executed(t1_newValue,2,c3).
    executed(t1_dropReq,3,c3).executed(t1_tune,4,c3).

    examples:-
    happens(start,0,c1), req_happens(t1_tune,1,c1),
    not happens(t1_tune,3,c2), not happens(t1_tune,4,c3).
```

At the beginning, as no domain specific properties are included in the background knowledge, all event transitions are by default maybe transitions. Hence in the stable model of the EC program we have *maybe(e,p,c)* for every combination of event e, position p and trace c given in the language. Furthermore, the model contains *happens(e,p,c)* for those events where also *executed(e,p,c)* has been added to the narrative of the background knowledge. In other words, before the learning, the background knowledge entails that all executed events happen as maybe events. However, the examples state that last *t1_tune* executed in traces $c2$ and $c3$ should not occur and that the same event is required to happen in trace $c1$. So the given background knowledge does not entail the given examples. The ILP task then computes the following set of hypotheses:

```
 required(t1_tune,X1,X2):-holds_at(f_start,P,C).
proscribed(t1_tune,X1,X2):-holds_at(tuning,P,C).                        (12)
                           not holds_at(waitingDropAck_t1,P,C).
```

The above rules are translated back into the safety properties

$$\begin{array}{l} \mathsf{G}(start \rightarrow \mathsf{X}\ (t1\_tune)) \\ \mathsf{G}((Tuning\_t1 \wedge WaitingDropAck\_t1) \rightarrow \mathsf{X}\ \neg(t1\_tune)) \end{array} \tag{13}$$

The MTS generated from the conjunction of learned assertions in (13) is shown in Figure 2. It is easy to show that the refined model is a strong refinement of the initial model given in Figure 1 since every possible transition in the refined MTS is a possible transition in the initial MTS, and every required transition in M, in this case just the single transition *start* from the initial state, is also preserved in the refined model.

5 Discussion

In our present approach, we have focused on learning universal properties that force one required transition from states that satisfy the properties' conditions. The choice to adopt this semantics was inspired by existing work on synthesising MTSs that satisfy properties universally and existentially. Weaker semantics, whereby there can be several required

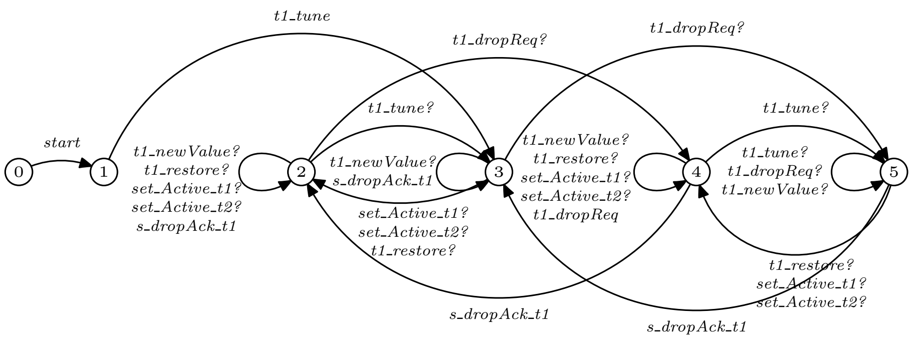

Figure 2 A refined MTS N for the Philip TV set

transitions from a single state to satisfy existential properties, can be modelled in our EC programs by adding an *executed* literal to the body of the definition of *req_happens* in (3) and dropping the constraint (8).

In defining our refinement, we have considered the MTSs that are generated from the conjunction of a given set of formulae instead of constructing an MTS for each formula and merging the single MTS models as proposed in [5]. Though only argued in [5] to be equivalent fro the case of safety properties and MTSs with the same alphabet, we have now verified through our refinement approach that this is indeed the case. We have also compared our work with techniques that synthesise MTS from scenarios directly [13] and found that such approaches do not support the use of negative scenarios which are crucial in our case to avoid over generalising the learned conditions.

In [1], we have presented an approach for detecting and resolving incompleteness in operational specifications using ILP. The this previous work the system models are Labelled Transition Systems which are less expressive than MTS as they are based on a completion assumption by which the system behaviour is strictly classified as either proscribed or required. The properties that were learned aimed at pruning undesirable traces from the initial Labelled Transition System. In this paper, the learning task is more general as it prunes traces where current possibly transitions should instead be proscribed, and also forces possible traces to be required. Both sets Δ_r and Δ_p of a given MTS are therefore consistently refined, whilst preserving the refinement relation between the MTS of the given description and that of the refined specifications.

6 Conclusion and Future Work

The overall aim of the work presented in this paper is to provide a formal and tool-supported approach for incremental software development through modal refinement by means of inductive learning. In brief, we have described the use of EC logic programs and ILP for computing strong refinements of MTSs from given event traces. We have shown how the EC language can sufficiently capture the different notions of the system models and how non-monotonic learning preserves the conditions of a strong refinement. We have argued that our computed hypothesis characterise a set of implementations of the original MTS.

As part of our future work, we intend to extend the approach to a wider class of safety properties, both as initial partial system descriptions and as learned properties, and to other forms of refinements such as weak refinement. This will enhance the applicability of our methodology to problems where the incremental refinement requires extending the given alphabet of events as new descriptions become available. We would also like to explore the use of ILP to reason and refine non-deterministic MTSs, so fully exploiting the benefit of the EC representation.

Acknowledgments

We acknowledge financial support for this work from ERC project PBM - FIMBSE (No. 204853). Additionally, we would like to thank Rob Miller for initial discussions and Oliver Ray for providing a prototype implementation of XHAIL.

References

1 D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel. Learning operational requirements from goal models. In *Proc. of 31st Intl. Conf. on Soft. Eng.*, pages 265–275, 2009.

2 D. Alrajeh, O. Ray, A. Russo, and S. Uchitel. Using abduction and induction for operational requirements elaboration. *J. of Applied Logic*, 7(3):275 – 288, 2009.

3 N. D'Ippolito, D. Fischbein, M. Chechik, and S. Uchitel. MTSA: The Modal Transition System Analyser. In *Proc. of 23rd Intl. Conf. on Auto. Soft. Eng.*, pages 475 –476, 2008.

4 D. Fischbein, V. Braberman, and S. Uchitel. A sound observational semantics for modal transition systems. In *Proc. of the 6th Intl. Colloquium on Theoretical Aspects of Comp.*, pages 215–230, 2009.

5 D. Fischbein and S. Uchitel. On correct and complete strong merging of partial behaviour models. In *Proc. of 16th Intl. Symp. on Foundations of Soft. Eng.*, pages 297–307, 2008.

6 M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R.A. Kowalski and K. Bowen, editors, *Proc. of 5th Intl. Conf. on Logic Programming*, pages 1070–1080, 1988.

7 D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *Proc. 11th ACM SIGSOFT Symp. on Foundations Soft. Eng.*, pages 257 – 266, 2003.

8 K. Larsen, U. Nyman, and A. Wasowski. On modal refinement and consistency. In *Proc. of 18th Intl. Conf. on Concurrency Theory*, pages 105–119, 2007.

9 K.G. Larsen and B. Thomsen. A modal process logic. In *Proc. of 3rd Annual Symp. on Logic in Computer Science*, pages 203 –210, 1988.

10 S.H. Muggleton. Inverse Entailment and Progol. *New Generation Comp. , Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.

11 O. Ray. Nonmonotonic abductive inductive learning. *J. of Applied Logic*, 7(3):329–340, 2009.

12 M. Shanahan. The event calculus explained. In *Artificial Intelligence Today: Recent Trends and Developments*, volume 1600 of *LNCS*, pages 409–430. 1999.

13 S. Uchitel, G. Brunet, and M. Chechik. Synthesis of partial behavior models from properties and scenarios. *Trans. on Soft. Eng.*, 35:384–406, 2009.

14 R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. *Computer*, 33:78–85, 2000.