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 / 3**
Title of Deliverable:	**Reasoning about Triggered Scenarios in Logic Programming**

Contractual Date of Delivery to the CEC:	**31-Mar-2015**
Actual Date of Delivery to the CEC:	**31-Mar-2015**
Organisation name of lead contractor for this deliverable:	**IMP**
Author(s):	**D. Alrajeh, R. Miller, A. Russo, S. Uchitel**
Participants(s):	**IMP,UBA**
Work package contributing to the deliverable:	**WP5**
Nature:	**R**
Dissemination Level:	**Public**
Total number of pages:	**20**
Start date of project:	1 Oct. 2011 **Duration:** 48 month

Abstract:

This document presents a logic programming approach, based on the Event Calculus (EC), for representing and reasoning about triggered scenarios (TS), an expressive dialect of message sequence charts widely employed in software requirements engineering to specify system behaviour. We introduce a sound translation for triggered scenarios into an EC-based Answer Set Programming (ASP) representation, and demonstrate how this formalisation allows for the use of ASP as an alternative verification method capable of overcoming known limitations of current TS analysis methods.

Note:

This deliverable is based on work presented at the International Conference on Logic Programming 2013.

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

Contents

1 Introduction

Requirements Engineering is the set of activities concerned with identifying and communicating the purpose of a software-intensive system, and the contexts in which it will be used. The elicitation and analysis of stakeholders' requirements for a system's behaviour are of paramount importance. To facilitate the direct involvement of stakeholders, requirements are often conveyed through *sequence charts* — intuitive narrative-style descriptions of desirable and undesirable system behaviours — for which *UML interaction diagrams* and in particular *message sequence charts* [**?**] are a widely accepted notation. *Triggered Scenarios* (TS) [**?**, **?**] are an expressive variant of sequence charts that allow the user to differentiate between the circumstances (or *triggers*) under which particular system behaviours should be facilitated. and the resultant behaviours themselves. They also allow a differentiation between behaviours that should be required after a trigger, described with *universal triggered scenarios* (uTS), and behaviours that should just be made possible within the system-to-be, described with *existential triggered scenarios* (eTS). Though frameworks supporting the automated analysis of TSs exist (e.g. [**?**]), they are 'incomplete' with respect to the behaviours they aim to cover.

In this paper we show how Event Calculus (EC), a standard logic-based AI framework for reasoning about actions and change, can be extended to provide a representation for a collection of TSs, and show how this representation corresponds to the existing semantics for TSs, which is expressed in terms of *computation trees* and *state functions*. We also demonstrate how the formalisation allows for the use of Answer Set Programming (ASP) solvers to perform various verification tasks on TSs.

2 Motivating Example

Consider the requirements engineering task of eliciting behaviour for an Air Traffic Control (ATC) system in which flight plans, radar information and aircraft must be coordinated to support flexible, safe passage through airspace [**?**]. Assume that we have been provided with the two examples of expected system behaviour in TS notation shown in Fig. 1. Intuitively, their intended meanings are as follows. Fig. 1.A indicates that if the ATC receives a request to change a flight path, approves it, and then receives a signal from the radar, then the ATC must send instructions to the aircraft and the radar must scan for other flights before any other event in the scenario's "scope" occurs. In other words, *send_instructions* and *scan* must occur before further occurrences of *request_change*, *approve* or *send_signal*. Fig. 1.B states that if the ATC receives details of a plan and then a radar signal, and afterwards the condition labelled [l_4] is true, then the ATC must update details of the approved plan before *send_details*, *send_signal* or changes to *Change_Requested* or *Approved* happen again. Because *send_instructions* and *scan* are not part of the scope of the second TS while *update_details* is not part of the scope of the first, the two TSs may be interleaved. For example, system executions would be possible in which *request_change*, *approve*, *send_details*, *send_signal* occur in turn, followed either by *send_instructions*, *scan* and *update_details*, or by *update_details*, *send_instructions* and *scan*.

In [?], an automatic translation of TSs such as these into modal transition systems (MTS) [?] is provided, and a model checker [?] can be used to automatically verify consistency of TSs as well as validity of temporal properties in possible system *implementations* that satisfy the TSs. However, the translation is incomplete as it relies on MTS conjunction (MTS are known not to be closed under conjunction) and hence the MTS model checker may return unsound results. In this particular example, we may conclude that the specification is inconsistent with an additional safety requirement stating that *send_instructions* may not be followed immediately by *update_details*. This is because MTS cannot model precisely what is specified in Fig. 1: that either order for *send_instructions* and *update_details* would be acceptable as a response to the triggering sequence. The best approximation that can be made using MTS is to assume one particular order must be present in the final system. In contrast, the EC-based techniques presented in this paper allow for TSs to merge in a correct way that preserves the intended meaning of each component.

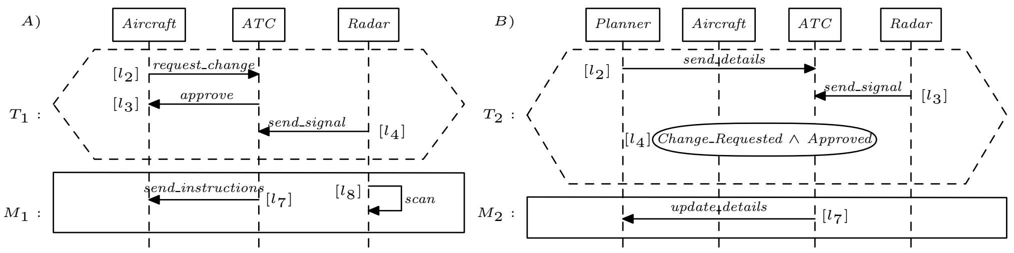

Figure 1: *Triggered Scenarios A) uTS_SendInstructionsAndScan* and *B) uTS_UpdateDetails*.

3 Triggered Scenarios

We identify both the TSs in Fig. 1 as uTSs (not eTSs) because their *main charts*, the regions M_1 and M_2, are enclosed in continuous (not dotted) lines. In a TS, the region T above M is the *trigger*. Intuitively, a uTS means that if a part of an execution trace matches a *linearisation* (compatible total ordering) of T's actions, then all possible continuations of that part must start by matching a linearisation of the main chart M. An eTS means that if a part of an execution trace matches a linearisation of T, then there must be at least one possible continuation of that part that starts by matching a linearisation of M. Moreover, for both uTSs and eTSs, every linearisation of M must be a possible continuation of any given linearistion of T.

In Fig. 1.B, the uTS has four vertical dotted *lifelines* (*Planner*, *Aircraft*, *ATC* and *Radar*) that represent different components of an ATC system. The arrows represent *actions* (e.g. messages) between lifelines, and the labels *send_details*, *send_signal* and *update_details* identify the action type. *Change_Requested* and *Approved* are *fluents* used to compose conditions (*fluent formulae*) that are true where they appear in the trigger. Points where actions or conditions meet lifelines are called *locations*. For explanatory convenience, locations have labels (e.g., “$[l_2]$”)[1]. In

[1]These labels are not part of the TS itself, and the indices on the labels are arbitrary with no semantic implication.

triggered scenarios, time flows from top to bottom along each lifeline, independently save for the constraints implied by the positions of the locations on each line. For example, in Fig. 1.A *send_signal* must occur after *approve* because $[l_4]$ is below $[l_3]$ on *ATC*'s lifeline. But *scan* can occur either before or after *send_instructions*. The fluent formula *Change_Requested* $\land$ *Approved* in Fig. 1.B must be true immediately after the occurrence of *send_signal*. In addition to the diagrams, a TS specification includes background information on (i) the initial truth values of all fluents and which actions change these values, and (ii) which actions are relevant to each TS and therefore can only occur where (if at all) they appear in the diagram (this set of actions is called the *scope* of the TS). To provide a semantics, TSs have been given a symbolic representation [**?**] in terms of *labelled partial orders with conditions* (LPOCs). We summarise this next. We assume a finite, universal set *Act* of action labels (for our example {*approve*, *request_change*, *send_signal*, *send_instructions*, *scan*, *send_details*, *update_details*, *change_ack*}).

Definition 1. A *fluent* F is a triple $\langle I_F, T_F, Init_F\rangle$ such that $I_F \cup T_F \subseteq Act$, $I_F \cap T_F = \emptyset$ and $Init_F \in \{\top, \bot\}$. I_F and T_F are the set of *initiating* and *terminating* actions respectively of F, and $Init_F$ is the *initial value*. Given $\Sigma \subseteq Act$, F is *defined over the alphabet* Σ if $I_F \cup T_F \subseteq \Sigma$.

We assume a finite, universal set *Fls* of fluent labels each of which uniquely identifies a fluent. In our example, *Fls* = {*Change_Requested*, *Approved* }, where *Approved* = $\langle$\{*approve*\}, \{*update_details*\}, $\bot\rangle$ and *Change_Requested* = $\langle$\{*request_change*\}, \{*change_ack*\}, $\bot\rangle$. For ease of exposition, we restrict fluent formulae to be conjunctions of fluent literals, although our approach is easily generalisable to disjunctions of such formulae as well:

Definition 2 (Fluent Propositional Logic (FPL) Formula)**.** Let $\mathscr{F}$ be the set of all fluents defined over $\Sigma \subseteq Act$ that have a fluent label in *Fls*. A fluent propositional logic (FPL) formula over Σ is an expression of the form $\phi_1 \land \ldots \land \phi_n$, where $n \geq 1$ and each ϕ_i is either $\mathtt{f}$ or $\neg\mathtt{f}$ for some $\mathtt{f} \in \mathscr{F}$.

In the following, $l \prec l'$ if and only l' appears immediately after l on a TS instance line. For example, in Fig. 1.A $l_2 \prec l_3$ but $l_2 \nprec l_4$.

Definition 3 (LPOC)**.** A *labelled partial order with conditions* (LPOC) is a tuple $\langle L, \prec, \lambda, \Sigma, \Psi\rangle$ where

- L is a finite set of *locations*,
- $\prec \subseteq L \times L$ is an antisymmetric, irreflexive, antitransitive relation such that $\prec \subseteq \leq$ for at least one total ordering $\leq$ over L,
- $\Sigma \subseteq Act$, and Ψ is a set of FPL formulae such that each fluent appearing in a formula in Ψ is defined over the alphabet Σ, and
- $\lambda : L \to \Sigma \cup \Psi$.

λ is referred to as the *labelling function*, and Σ as the *scope* or *alphabet* of the LPOC. L^Σ refers to the set of action label locations $\{l \mid l \in L \text{ and } \lambda(l) \in \Sigma\}$, and L^Ψ to the set of fluent label locations $\{l \mid l \in L \text{ and } \lambda(l) \in \Psi\}$.

In a TS, the trigger and main chart are each represented as LPOCs, where the main chart LPOC has an empty Ψ.

Definition 4 (Triggered Scenario (uTS/eTS)). A *universal triggered scenario* (uTS) is a tuple $U = \Box(T, M, \Sigma)$ and an *existential triggered scenario* (eTS) is a tuple $E = \Diamond(T, M, \Sigma)$, where the *trigger* T and the *main chart* M are LPOCs with alphabet Σ.

For example, the uTS of Fig. 1.A is $\Box(T_1, M_1, \Sigma_1)$ where
$\Sigma_1 = \{\mathit{request\_change}, \mathit{approve}, \mathit{send\_signal}, \mathit{send\_instructions}, \mathit{scan}\}$,
$T_1 = \langle\{l_2, l_3, l_4\}, \{(l_2, l_3), (l_3, l_4)\}, \{\lambda(l_2) = \mathit{request\_change}, \lambda(l_3) = \mathit{approve},$
$\lambda(l_4) = (\mathit{send\_signal})\}, \Sigma_1, \{\}\rangle$ and
$M_1 = \langle\{l_7, l_8\}, \emptyset, \{\lambda(l_7) = \mathit{send\_instructions}, \lambda(l_8) = \mathit{scan}\}, \Sigma_1, \{\}\rangle$.

The semantics of TSs is expressed in [?] in terms of *words* and *computation trees*. A *word* is a (possibly empty or infinite) sequence of actions in *Act*. For any $A \subseteq \mathit{Act}$, A^* denotes the set of all finite words constructed from A, including the *empty* word ϵ. For example (*send_details send_signal*) and (*send_signal send_details*) are both members of $\{\mathit{send\_signal}, \mathit{send\_details}\}^*$. For a given $\Sigma \subseteq A$, the *projection* of a word $w \in A^*$ onto the set Σ, denoted $w|_\Sigma$, is the word constructed from w by removing actions that are not included in Σ without otherwise altering the ordering of w. Note that $\epsilon|_\Sigma = \epsilon$. If w, u and v are words such that $w = uv$, then u is called a *prefix* of w. A computation tree is a transition system represented as a tree structure in which each node corresponds to a word, the root being the empty word.

Definition 5 (Computation Tree). A *computation tree* is a transition system $(V, A, \Delta, v_\epsilon)$, where $A \subseteq \mathit{Act}$, V is a set of *nodes* each of which is uniquely indexed by a single element of A^* (so that the set of node indices is a subset of A^*), v_ϵ is the *initial node* (indexed with the empty word ϵ), and Δ is a transition relation, $\Delta \subseteq V \times A \times V$, such that $(v_w, \alpha, v_{w'}) \in \Delta$ if and only if $w' = w\alpha$.

Note from Def. 5 that any set $V^I \subseteq A^*$ of node indices identifies a unique computation tree. Moreover, if for every $w \in V^I$ there is an $\alpha, \beta \in A$ and $w' \in V^I$ such that $w\alpha \in V^I$ and (for $w \neq \epsilon$) $w = w'\beta$, then the computation tree associated with V^I is a classical tree structure with root node v_ϵ such that every path from v_ϵ is infinite. Such computation trees are called *deadlock-free*. A *branch* of a computational tree is a (possibly infinite) sequence $b = (v_w, \alpha, v_{w\alpha})(v_{w\alpha}, \alpha', v_{w\alpha\alpha'})\ldots$ of transitions in Δ, that can start at any node (not just v_ϵ).

To define when a computation tree satisfies a given triggered scenario [?] defines the notions of *state function*, *linearisation* and satisfaction of an LPOC by a word. A *state function Fls*, denoted ζ, captures the valuation of fluents at (the end of) a given word. In other words, given a fluent $F \in \mathit{Fls}$ and a word w, the state function at w, $\zeta(w)$, maps the fluent to (i) false (respectively true) if one of its terminating (respectively initiating) actions appears in w after all (if any) occurrences of its initiating (respectively terminating) actions, and (ii) its initial value otherwise.

A word w is a *linearisation* of an LPOC if the location labels L^Σ can be (re)indexed into a sequence $l_0, ..., l_n$ such that, for all $1 \le i, j \le n$, if $l_i \prec l_j$ then $i < j$, and the sequence matches w (i.e. $w = \lambda(l_0)...\lambda(l_n)$). For instance, the word (*scan send_instructions*) is a linearisation of M_1 in Fig. 1.A. The FPL formula ϕ *is at prefix* w' *of* w *with respect to an LPOC* iff there is a location

$l' \in L^\Psi$ such that $\lambda(l') = \phi$ and either (i) for some $0 \le k \le n$, $l_k \prec^\Psi l'$ and $w' = \lambda(l_0)...\lambda(l_k)$, or (ii) $w' = \epsilon$ and there is no $l \in L^\Sigma$ such that $l \prec^\Psi l'$.

A word w satisfies an LPOC C, written $w \models C$, if there is a decomposition $w = uv$ such that (i) $v|_\Sigma$ is a linearisation of C, and (ii) for every prefix v' of v and FPL formulae ϕ such that ϕ is at prefix $v'|_\Sigma$ of $v|_\Sigma$ with respect to C, the FPL ϕ is true under the interpretation $\zeta(uv')$.

Definition 6 (Satisfaction of a eTS)**.** A computation tree *satisfies* the eTS $E = \Diamond(T, M, \Sigma)$ iff (i) it is deadlock-free and (ii) whenever a word w defined by a finite branch b that starts at the tree's initial node is such that $w \models T$, then for every linearisation z of M there exists a branch b' starting at the ending node of b that defines a word w' such that $w'|_\Sigma = z$.

Definition 7 (Satisfaction of a uTS)**.** A computation tree *satisfies* the uTS $U = \Box(T, M, \Sigma)$ iff (i) it satisfies the eTS $E = \Diamond(T, M, \Sigma)$ and (ii) whenever a word w defined by a finite branch b that starts at the tree's initial node is such that $w \models T$, then any infinite branch b' that starts at the ending node of b defines a word of the form uv for some $u \in Act^*$ such that $u|_\Sigma$ is a linearisation of M.

Finally, we say that a computation tree satisfies a set of TSs if it satisfies each member of the set.

4 An Event Calculus Formalisation of Triggered Scenarios

The EC formalisation of Triggered Scenarios is in ASP and models time as a set of parallel *runs*, each representing an execution trace (i.e. a linear sequence of actions) of a system. The axiomatisation is based on a version of the EC in [?], customised for the special case where the domain is deterministic and the initial state is completely specified. This uses a sort `action` for actions (with variables $\mathtt{A}, \mathtt{A}_1, \mathtt{A}_2, \ldots$), a sort `fluent` for fluents (with variables $\mathtt{F}, \mathtt{F}_1, \mathtt{F}_2, \ldots$), a sort `run` for runs (with variables $\mathtt{R}, \mathtt{R}_1, \mathtt{R}_2, \ldots$) and a sort `position` for positions within a run (with variables $\mathtt{P}, \mathtt{P}_1, \mathtt{P}_2, \ldots$). The main predicates are `happens(A,R,P)`, `holdsAt(F,R,P)`, `initiates(A,F,R,P)` and `terminates(A,F,R,P)`. It is convenient to also define an auxiliary predicate $\mathtt{clipped}(\mathtt{R}, \mathtt{P}_1, \mathtt{F}, \mathtt{P}_2)$ which means that, in a given run `R`, an action occurs which terminates `F` between positions $\mathtt{P}_1$ and $\mathtt{P}_2$. The corresponding definitions are:

$$\mathtt{clipped}(\mathtt{R},\mathtt{P}_1,\mathtt{F},\mathtt{P}_2) \leftarrow \mathtt{happens}(\mathtt{A},\mathtt{R},\mathtt{P}),\ \mathtt{P}_1 \le \mathtt{P},\ \mathtt{P} < \mathtt{P}_2,\ \mathtt{terminates}(\mathtt{A},\mathtt{F},\mathtt{R},\mathtt{P}). \tag{EC1}$$

$$\mathtt{holdsAt}(\mathtt{F},\mathtt{R},\mathtt{P}_2) \leftarrow \mathtt{happens}(\mathtt{A},\mathtt{R},\mathtt{P}_1),\ \mathtt{P}_1 < \mathtt{P}_2,\ \mathtt{initiates}(\mathtt{A},\mathtt{F},\mathtt{R},\mathtt{P}_1),\ \mathtt{not}\ \mathtt{clipped}(\mathtt{R},\mathtt{P}_1,\mathtt{F},\mathtt{P}_2). \tag{EC2}$$

$$\mathtt{holdsAt}(\mathtt{F},\mathtt{R},\mathtt{P}_2) \leftarrow \mathtt{holdsAt}(\mathtt{F},\mathtt{R},\mathtt{P}_1),\ \mathtt{P}_1 < \mathtt{P}_2,\ \mathtt{not}\ \mathtt{clipped}(\mathtt{R},\mathtt{P}_1,\mathtt{F},\mathtt{P}_2). \tag{EC3}$$

These axioms formalise a commonsense law of inertia: in any run `R`, a fluent that is either initially true (resp. false) or has been initiated (resp. terminated) by an action occurrence continues to hold (resp. not to hold) until a terminating (resp. initiating) action occurs. Information about which actions affect which fluents is provided by domain-dependent axioms for the predicates `initiates` and `terminates`, together with information about which fluents are initially true.

As we shall see, to capture the semantics of TSs we need to be able to express that two runs contain the same sequence of actions up to a given position. We do this with a predicate

$\texttt{sameHistory}(\texttt{R}_1, \texttt{R}_2, \texttt{P})$ defined as follows:

$\texttt{sameHistory}(\texttt{R}_1, \texttt{R}_2, \texttt{P}) \leftarrow \texttt{not differentHistory}(\texttt{R}_1, \texttt{R}_2, \texttt{P}).$ (EC4)

$\texttt{differentHistory}(\texttt{R}_1, \texttt{R}_2, \texttt{P}) \leftarrow \texttt{P}_0 < \texttt{P}, \texttt{happens}(\texttt{A}, \texttt{R}_1, \texttt{P}_0), \texttt{ not happens}(\texttt{A}, \texttt{R}_2, \texttt{P}_0).$ (EC5)

Furthermore, because runs represent system executions, exactly one action must occur at each position of each run. This is captured by the integrity constraints (EC6) and (EC7), with the predicate occurs providing the necessary existential quantification of the action argument:

$\leftarrow \texttt{happens}(\texttt{A}_1, \texttt{R}, \texttt{P}), \ \texttt{happens}(\texttt{A}_2, \texttt{R}, \texttt{P}), \ \texttt{A}_1 \neq \texttt{A}_2.$ (EC6)

$\leftarrow \texttt{not occurs}(\texttt{R}, \texttt{P}).$ (EC7)

$\texttt{occurs}(\texttt{R}, \texttt{P}) \leftarrow \texttt{happens}(\texttt{A}, \texttt{R}, \texttt{P}).$ (EC8)

We define the EC formalisation of a TS-based system specification by describing a translation function Γ[...] for each of its components. $\Gamma[Act]$ is simply $\{\texttt{action}(\texttt{a}) | \texttt{a} \in Act\}$. $\Gamma[Fls]$ includes a set of atoms of the form fluent(f) for each $\texttt{f} \in Fls$, a set of atoms $\texttt{holdsAt}(\texttt{f}, \texttt{R}, \texttt{0})$, for each fluent f that is initially true (i.e., $Init_{\texttt{f}} = \top$), and, for each f, atoms $\texttt{initiates}(\texttt{a}, \texttt{f}, \texttt{R}, \texttt{P})$ for each $\texttt{a} \in I_{\texttt{f}}$ and atoms $\texttt{terminates}(\texttt{a}, \texttt{f}, \texttt{R}, \texttt{P})$ for each $\texttt{a} \in T_{\texttt{f}}$. So in our example $\Gamma[Fls]$ includes initiates(request_change, change_Requested, R, P), terminates(change_ack, change_Requested, R, P), etc.

Each TS in a system specification has an overall name, and names for its trigger and main chart, captured by the sort predicates trigger_scenario, trigger and main. We use the predicate scope to state which actions are in each TS's scope, the predicate is_in to state that a trigger or main chart belongs to a particular TS and the predicate uni to state that the TS is universal. For instance, our example includes trigger_scenario(uTS_UpdateDetails), uni(uTS_UpdateDetails), trigger(t2), main(m2), is_in(t2, uTS_UpdateDetails), etc, along with atoms scope(send_details, uTS_UpdateDetails), etc.

The translation of an FPL formula ϕ is with respect to a given position and run and is expressed in terms of holdsAt. So given a fluent f, fluent literals $\phi_1, \ldots, \phi_n$, a position P and a run R, $\Gamma[\texttt{f}, \texttt{R}, \texttt{P}]$ is $\texttt{holdsAt}(\texttt{f}, \texttt{R}, \texttt{P})$, $\Gamma[\neg\texttt{f}, \texttt{R}, \texttt{P}]$ is $\texttt{not holdsAt}(\texttt{f}, \texttt{R}, \texttt{P})$, and $\Gamma[\phi_1 \wedge \ldots \wedge \phi_n, \texttt{R}, \texttt{P}]$ is $\Gamma[\phi_1, \texttt{R}, \texttt{P}], \ldots, \Gamma[\phi_n, \texttt{R}, \texttt{P}]$.

To translate the LPOC $C = \langle L, \prec, \lambda, \Sigma, \Psi \rangle$ we need some extra notation related to $\prec$. The relation $\prec^{\Sigma}$ is analagous to $\prec$ but completely disregarding any locations of fluent formulae: $l \prec^{\Sigma} l'$ iff $l, l' \in L^{\Sigma}$ and either (i) $l \prec l'$, or (ii) there is a set $\{x_0, ..., x_n\} \subseteq L^{\Psi}$, such that $l \prec x_0$, $x_n \prec l'$, and $x_{i-1} \prec x_i$ for each $1 \leq i \leq n$. The relation $\prec^{\Psi}$ identifies the location of the last action (if there is one) before a given fluent formula along each of the instance time lines in C. Formally, $l \prec^{\Psi} l'$ iff $l \in L^{\Sigma}$, $l' \in L^{\Psi}$ and either (i) $l \prec l'$, or (ii) there is a set $\{x_0, ..., x_n\} \subseteq L^{\Psi}$, such that $l \prec x_0$, $x_n \prec l'$, and $x_{i-1} \prec x_i$ for each $1 \leq i \leq n$. Thus for the trigger in Fig. 1.B, $\prec^{\Sigma} = \{(l_2, l_3)\}$ and $\prec^{\Psi} = \{(l_2, l_4), (l_3, l_4)\}$.

Our translation of LPOCs into EC formulae converts locations to position variables whose ordering is appropriately constrained by consideration of $\prec^{\Sigma}$ and $\prec^{\Psi}$:

Definition 8 (Translation of LPOCs)**.** Let the LPOC $\texttt{c} = \langle L, \prec, \lambda, \Sigma, \Psi \rangle$ be either the trigger or the main chart of a TS s. Without loss of generality, assume the locations in L are (arbitrarily) indexed using all but the first and last of a consecutive sequence $start, ..., k, end$ of natural numbers, i.e. $L = \{l_{start+1}, ..., l_k\}$. The translation makes use of variables $\texttt{P}_{start}, \ldots, \texttt{P}_k, \texttt{P}_{end}$ of sort `position`. Let $\overrightarrow{\mathtt{P}}$ be a shorthand for the sequence of variables $\mathtt{P}_{start+1}, \ldots, \mathtt{P}_k$. The translation $\Gamma[\mathtt{c}]$ of `c` is given by the following two clauses[2]:

$$\begin{aligned}
&\mathtt{lpoc}(\mathtt{c}, \mathtt{P}_{start}, \mathtt{P}_{end}, \mathtt{R}) \leftarrow \mathtt{P}_{start} < \mathtt{P}_{end}, \Gamma_{init}[\mathtt{c}], \Gamma_{<}[\mathtt{c}], \Gamma_{minmax}[\mathtt{c}], \Gamma_{hap}[\mathtt{c}], \Gamma_{holds}[\mathtt{c}],\\
&\qquad\qquad \mathtt{not\ scoped\_action\_happens}(\mathtt{c}, \mathtt{P}_{start}, \mathtt{P}_{end}, \mathtt{R}, \overrightarrow{\mathtt{P}}).\\
&\mathtt{scoped\_action\_happens}(\mathtt{c}, \mathtt{P}_{start}, \mathtt{P}_{end}, \mathtt{R}, \overrightarrow{\mathtt{P}}) \leftarrow \mathtt{action}(\mathtt{A}), \mathtt{is\_in}(\mathtt{c}, \mathtt{s}), \mathtt{scope}(\mathtt{A}, \mathtt{s}),\\
&\qquad \mathtt{position}(\mathtt{PA}), \Gamma_{noteq}[\mathtt{c}, \mathtt{PA}],\ \mathtt{P}_{start} \leq \mathtt{PA}, \mathtt{PA} < \mathtt{P}_{end}, \mathtt{happens}(\mathtt{A}, \mathtt{R}, \mathtt{PA}).
\end{aligned}$$

where:

- $\Gamma_{init}[\mathtt{c}]$ is the conjunction of $\{\Gamma[\lambda(l_m), \mathtt{R}, \mathtt{P}_{start}] \mid l_m \in L^{\Psi}$ and $\neg\exists\, l_j \in L$ s.t. $l_j \prec^{\Psi} l_m \}$
- $\Gamma_{<}[\mathtt{c}]$ is the conjunction of $\{(\mathtt{P}_j < \mathtt{P}_m) \mid l_j \prec^{\Sigma} l_m\}$
- $\Gamma_{minmax}[\mathtt{c}]$ is the conjunction of $\{(\mathtt{P}_{start} \leq \mathtt{P}_j), (\mathtt{P}_j < \mathtt{P}_{end}) \mid l_j \in L^{\Sigma}\}$
- $\Gamma_{hap}[\mathtt{c}]$ is the conjunction of $\{\mathtt{happens}(\lambda(l_j), \mathtt{R}, \mathtt{P}_j) \mid l_j \in L^{\Sigma}\}$
- $\Gamma_{holds}[\mathtt{c}]$ is the conjunction of $\{\Gamma[\lambda(l_m), \mathtt{R}, succ(\mathtt{P}_j)] \mid l_j \prec^{\Psi} l_m\}$
- $\Gamma_{noteq}[\mathtt{c}, \mathtt{PA}]$ is the conjunction of $\{\mathtt{PA} \neq \mathtt{P}_j \mid l_j \in L^{\Sigma}\}$

$\Gamma_{\mathcal{LPOC}}$ denotes the set of all clauses $\Gamma[\mathtt{c}]$ for some trigger or main chart `c`. To translate main charts we need a representation of each linearisation, identified by an *l-ordering*. We say that $\prec_+$ is a l-ordering of $\prec^{\Sigma}$ if (i) it is antisymmetric, irreflexive and antitransitive, (ii) its reflexive and transitive closure (denoted $\prec_+^{rtc}$) is a total order over L^{Σ}, and (iii) $\prec^{\Sigma} \subseteq \prec_+^{rtc}$. Thus for the main chart of the TS in Fig. 1.A there are two l-orderings: $\{(l_7, l_8)\}$ and $\{(l_8, l_7)\}$. We use a sort `linearisation_id` for l-ordering identifiers $\mathtt{l}_0$, $\mathtt{l}_1$, etc and define a predicate `linearisation_of(l, m)` for each l-ordering of a main chart `m`.

Definition 9 (Translation of LPOC with respect to $\prec_+$)**.** Let $\mathtt{c} = \langle L, \prec, \lambda, \Sigma, \Psi\rangle$ be an LPOC and let $\prec_+$ be an l-ordering of $\prec^{\Sigma}$ with identifier `l`. As for Definition 8, assume $L = \{l_{start+1}, ..., l_k\}$. The translation makes use of variables $\mathtt{P}_{start}, \ldots, \mathtt{P}_k, \mathtt{P}_{end}$ of sort `position`. Let $\overrightarrow{\mathtt{P}}$ be a shorthand for the sequence of variables $\mathtt{P}_{start+1}, \ldots, \mathtt{P}_k$. Let $\Gamma_{\prec_+}[\mathtt{c}]$ be the conjunction of the set $\{(\mathtt{P}_j < \mathtt{P}_m) \mid l_j, l_m \in L^{\Sigma}$ and $l_j \prec_+ l_m\}$. The translation of `c` with respect to $\prec_+$, denoted $\Gamma[\mathtt{c}, \prec_+]$, is given by the following clause:

$$\begin{aligned}
&\mathtt{linearisation}(\mathtt{c}, \mathtt{l}, \mathtt{P}_{start}, \mathtt{R}) \leftarrow \Gamma_{\prec_+}[\mathtt{c}], \Gamma_{minmax}[\mathtt{c}], \Gamma_{hap}[\mathtt{c}], \qquad \text{(L1)}\\
&\qquad\qquad \mathtt{not\ scoped\_action\_happens}(\mathtt{c}, \mathtt{P}_{start}, \mathtt{P}_k, \mathtt{R}, \overrightarrow{\mathtt{P}}).
\end{aligned}$$

$\Gamma_{\mathcal{L}}$ denotes the set of all clauses $\Gamma[\mathtt{sm}, \prec_+]$ for some l-ordering $\prec_+$ of some main chart `sm`[3]. We next describe when a run `R` contains an occurrence of a main chart `SM` starting at position $\mathtt{P}_i$:

$$\mathtt{main\_chart}(\mathtt{SM}, \mathtt{P}_{start}, \mathtt{R}) \leftarrow \mathtt{P}_{start} < \mathtt{P}_{end}, \mathtt{lpoc}(\mathtt{SM}, \mathtt{P}_{start}, \mathtt{P}_{end}, \mathtt{R}). \qquad \text{(EC9)}$$

Recall that an eTS expresses that if a part of an execution trace of the system matches its trigger, then, for each main chart linearisation, there must exist a possible continuation of that trace that

[2]For simplicity some type predicates have been omitted from the body of these clauses.

[3]See Appendix A for an example of the EC translation from the triggered scenario Fig 1.A.

starts by matching the linearisation. Additionally, a uTS expresses that no other continuations of the trace are possible. These requirements are captured by the following constraints and clause:

$$\leftarrow \texttt{trigger(ST)},\texttt{main(SM)},\texttt{is\_in(ST,S)},\texttt{is\_in(SM,S)},\texttt{lpoc}(\texttt{ST},\texttt{P}_{start},\texttt{P}_{end},\texttt{R}), \quad \text{(EC10)}$$
$$\texttt{linearisation\_id(L)},\texttt{linearisation\_of(L,SM)},$$
$$\texttt{not exists\_linearisation\_with\_same\_history}(\texttt{SM},\texttt{L},\texttt{P}_{end},\texttt{R}).$$

$$\leftarrow \texttt{trigger(ST)},\texttt{main(SM)},\texttt{is\_in(ST,S)},\texttt{is\_in(SM,S)},\texttt{uni(S)}, \quad \text{(EC11)}$$
$$\texttt{lpoc}(\texttt{ST},\texttt{P}_{start},\texttt{P}_{end},\texttt{R}),\texttt{not main\_chart}(\texttt{SM},\texttt{P}_{end},\texttt{R}).$$

$$\texttt{exists\_linearisation\_with\_same\_history}(\texttt{SM},\texttt{L},\texttt{P},\texttt{R}_1) \leftarrow \texttt{main(SM)}, \quad \text{(EC12)}$$
$$\texttt{linearisation\_id(L)},\texttt{sameHistory}(\texttt{R}_1,\texttt{R}_2,\texttt{P}),\texttt{linearisation}(\texttt{SM},\texttt{L},\texttt{P},\texttt{R}_2).$$

To compensate for the finiteness of ASP, our representation includes an additional integrity constraint (EC13) below, stating that the end point of each run must match with a "system state" already reached further back in the same or another run. Additionally, from this other matching run position, progress must be made through each main chart that has already started. Informally, a system state is characterised by both the fluents that hold, and by the current position on each "pathway" through each TS, where a pathway is a linearisation of the trigger concatenated with a linearisation of the main chart. Each possible pathway position in each TS is identified with a unique constant (e.g. $\texttt{e}_{23}$) of sort `execution_state`. These notions are captured in the following constraint and accompanying clauses:

$$\leftarrow \texttt{max\_position(P)},\texttt{not has\_matching\_earlier\_point(R,P)}. \quad \text{(EC13)}$$

$$\texttt{has\_matching\_earlier\_point(R,P)} \leftarrow \texttt{run}(\texttt{R}_1),\texttt{position}(\texttt{P}_1),\texttt{P}_1 < \texttt{P},$$
$$\texttt{same\_Execution}(\texttt{R},\texttt{P},\texttt{R}_1,\texttt{P}_1),\texttt{same\_Fluents}(\texttt{R},\texttt{P},\texttt{R}_1,\texttt{P}_1),\texttt{progress\_from}(\texttt{R}_1,\texttt{P}_1).$$
$$\texttt{progress\_from}(\texttt{R}_1,\texttt{P}_1) \leftarrow \texttt{not non\_progressor\_at}(\texttt{R}_1,\texttt{P}_1).$$
$$\texttt{non\_progressor\_at}(\texttt{R}_1,\texttt{P}_1) \leftarrow \texttt{holds\_execution\_state}(\texttt{ES},\texttt{R}_1,\texttt{P}_1),$$
$$\texttt{trigger\_complete(ES)},\texttt{not progresses}(\texttt{ES},\texttt{R}_1,\texttt{P}_1).$$
$$\texttt{progresses}(\texttt{ES},\texttt{R}_1,\texttt{P}_1) \leftarrow \texttt{position}(\texttt{P}_2),\texttt{P}_1 < \texttt{P}_2,\texttt{not holds\_execution\_state}(\texttt{ES},\texttt{R}_1,\texttt{P}_2).$$
$$\texttt{same\_Execution}(\texttt{R},\texttt{P},\texttt{R}_1,\texttt{P}_1) \leftarrow \texttt{not different\_Execution}(\texttt{R},\texttt{P},\texttt{R}_1,\texttt{P}_1).$$
$$\texttt{different\_Execution}(\texttt{R},\texttt{P},\texttt{R}_1,\texttt{P}_1) \leftarrow \texttt{holds\_execution\_state(ES,R,P)},$$
$$\texttt{not holds\_execution\_state}(\texttt{ES},\texttt{R}_1,\texttt{P}_1).$$
$$\texttt{different\_Execution}(\texttt{R},\texttt{P},\texttt{R}_1,\texttt{P}_1) \leftarrow \texttt{not holds\_execution\_state(ES,R,P)}$$
$$\texttt{holds\_execution\_state}(\texttt{ES},\texttt{R}_1,\texttt{P}_1).$$
$$\texttt{same\_Fluents}(\texttt{R},\texttt{P},\texttt{R}_1,\texttt{P}_1) \leftarrow \texttt{not different\_Fluents}(\texttt{R},\texttt{P},\texttt{R}_1,\texttt{P}_1).$$
$$\texttt{different\_Fluents}(\texttt{R},\texttt{P},\texttt{R}_1,\texttt{P}_1) \leftarrow \texttt{holdsAt(F,R,P)},\texttt{not holdsAt}(\texttt{F},\texttt{R}_1,\texttt{P}_1).$$
$$\texttt{different\_Fluents}(\texttt{R},\texttt{P},\texttt{R}_1,\texttt{P}_1) \leftarrow \texttt{not holdsAt(F,R,P)},\texttt{holdsAt}(\texttt{F},\texttt{R}_1,\texttt{P}_1).$$

Space limits prevent us from giving a general recipe for generating `holds_execution_state` clauses from a TS specification. Instead, we give an example `holds_execution_state` clause for the execution state identified by the constant $\texttt{e}_{23}$ of the TS in Fig 1.A, where the run has passed l_2 and l_3 but not yet passed l_4. Note that in this example $\overrightarrow{\texttt{P}}$ denotes the sequence $\texttt{P}_1,\texttt{P}_2$.

$$\texttt{holds\_execution\_state}(\texttt{e}_{23},\texttt{R},\texttt{P}) \leftarrow \texttt{P}_{start} \le \texttt{P}_1,\texttt{P}_1 < \texttt{P}_2,\texttt{P}_2 \le \texttt{P},$$
$$\texttt{happens}(\texttt{request\_Change},\texttt{R},\texttt{P}_1),\texttt{happens}(\texttt{approve},\texttt{R},\texttt{P}_2),$$
$$\texttt{not scoped\_action\_happens}(\texttt{t1},\texttt{P}_{start},\texttt{P},\texttt{R},\overrightarrow{\texttt{P}}).$$

$\Gamma_{\mathcal{ES}}$ denotes the set of all `holds_execution_state` clauses for all execution states arising from all pathways in all TSs in the specification.

The full translation of a set of triggered scenarios is defined as follows.

Definition 10 (Translation of a Set of Triggered Scenarios)**.** Let $\mathcal{U}$ be a finite set of uTSs, $\mathcal{E}$ be a finite set of eTSs and $\mathcal{S} = \mathcal{U} \cup \mathcal{E}$ a given system specification. The EC-based answer set programming representation of $\mathcal{S}$ with respect to a bound N, denoted $\Gamma[\mathcal{S}, \mathtt{N}]$, is the set of clauses:

$$\{(\text{EC1}), \ldots, (\text{EC13})\} \cup \Gamma[Act] \cup \Gamma[Fls] \cup \Gamma_{\mathcal{LPOC}} \cup \Gamma_{\mathcal{L}} \cup \Gamma_{\mathcal{ES}}$$

together with clauses defining auxiliary and sort predicates, and the following aggregate:

```
N [happens(E,R,P) : action(E): run(R): position(P)] N.
```

The value of N depends on the type of verification task in hand. Examples of such tasks are given in Section 5. Any completeness result of our translation would have to be with respect to a lower bound for N, and we leave this for future work. In this paper we rely instead on the soundness of our translation, as stated in Proposition 1 below, and focus on corresponding verification tasks.

Definition 11. Let r be a run, p be a position, and $H = \{\mathtt{happens}(\mathtt{a}_0, \mathtt{r}, \mathtt{0}) \ldots \mathtt{happens}(\mathtt{a}_p, \mathtt{r}, \mathtt{p})\}$. The word w_H defined by H is the finite sequence $a_0, ..., a_p$.

Proposition 1. *Let $\mathcal{S} = \mathcal{U} \cup \mathcal{E}$ where $\mathcal{U}$ is a finite set of uTSs and $\mathcal{E}$ is a finite set of eTSs. Let Δ be an ASP solution of $\Gamma[\mathcal{S}, \mathtt{N}]$ for some N. Then there exists a computation tree CT that satisfies $\mathcal{S}$ such that, for each $H \subset \Delta$ which is of the form $\{\mathtt{happens}(\mathtt{a}_0, \mathtt{r}, \mathtt{0}) \ldots \mathtt{happens}(\mathtt{a}_p, \mathtt{r}, \mathtt{p})\}$ for some run r and position p, the word w_H defined by H is a node index of CT, and, for each fluent f, $\mathtt{holdsAt}(\mathtt{f}, \mathtt{r}, \mathtt{p}) \in \Delta$ if and only if $\zeta(w_H)(\mathtt{f}) = true$.*

5 Bounded verification in ASP

Verification in software engineering refers to the process of automatically checking whether a system satisfies some given desirable property. It takes as input a description D, specified in a formal or semi-formal language (e.g., scenario notation), and a property ψ typically expressed in a temporal logic formalism. The task is to check that the behaviour captured in the semantics of D satisfies ψ, denoted as $D \models \psi$, for some given notion of satisfiability. If the verification procedure finds (at least) one behaviour that violates the property, a system execution (either in the form of a trace or tree depending on whether the semantics of the property is of a linear or branching nature) is produced illustrating how such a violation may be reached.

In this paper, we are interested in checking whether a description D, comprised of a set of universal and existential TSs, satisfies a class of properties referred to as *safety properties* [**?**]. A safety property expresses the notion that no 'bad' action or state will ever happen or be reached respectively. We focus our attention on safety properties that can be violated by a single word in computation tree. A violation of a safety property ψ occurs when ψ is false at an index of a computation tree satisfying D. We demonstrate below how the ASP solver iclingo [**?**] may be used to detect vacuity and violations to single-state fluent properties. However, our approach can be generalised to handle other forms of temporal safety properties.

Vacuity Detection

Though formalisms such as TSs are useful in capturing conditional behaviour of a system, they are liable to being satisfied vacuously. A vacuous implementation satisfies a set of TSs by never satisfying any of the TSs' triggers. For instance, in the case of our running example, a computation tree which never executes the action *approve* would satisfy both scenarios in Fig. 1 according to the notion of satisfiability given in Definition 7. The problem with accepting implementations that vacuously satisfy TSs is that such models may conceal inconsistency between the TSs, e.g., where two scenarios with the same scope require incompatible main charts to be executed. Detecting vacuity involves verifying a computation tree that satisfies the TSs against the property "the trigger of each TS is never exhibited" [**?**], i.e., there is no index that satisfies the trigger. Hence a violation to this property is a run showing how the system may execute a trigger.

This task can be formulated in terms of ASP. Given a finite set of triggered scenarios $\mathcal{S} = \mathcal{U} \cup \mathcal{E}$, the task is to find solutions to the program $\Gamma[\mathcal{S}, \mathtt{N}]$ for some N in which there is at least one atom lpoc(s, $\mathtt{p}_1$, $\mathtt{p}_2$, r) that is true for each trigger s in the set. We therefore extend our ASP representation with the following aggregate constraint for each trigger t in a TS in $\mathcal{S}$:

$\leftarrow$ 0 [lpoc(t,P_1,P_2,R):position(P_1):position(P_2):run(R)] 0

The above aggregates ensure that for each trigger any ASP solution includes at least one run where that trigger is executed. If no solution is found then this means that the TSs cannot be satisfied non-vacuously and therefore the specification will need to be revised. For instance, suppose the triggered scenarios in Fig. 1 have been modified so that the action *send_details* in the trigger of Fig. 1.B no longer appears and so that both scenarios share the same scope, i.e. $\Sigma_1 = \Sigma_2 = \{$*approve*, *send_signal*, *request_change*, *change_ack*, *update_details*, *send_instructions*, *scan*$\}$. We refer to the modified scenarios as s_1 and s_2 and $\mathcal{S} = s_1 \cup s_2$. Running iclingo on the program $\{\Gamma[\mathcal{S}, \mathtt{N}]\}$ results in a number of ASP solutions including a solution that contains:

$H_1 = \{$ happens(approve,r1,0) happens(approve,r1,1)
happens(send_signal,r1,2) happens(send_signal,r1,3)
happens(change_ack,r1,4) happens(change_ack,r1,5)
happens(update_details,r1,6) happens(send_signal,r1,7)
happens(change_ack,r1,8)$\}$
$H_2 = \{$ happens(request_change,r2,0) happens(send_signal,r2,1)
happens(send_signal,r2,2) happens(approve,r2,3)
happens(change_ack,r2,4) happens(approve,r2,5)
happens(update_details,r2,6) happens(request_change,r2,7)
happens(update_details,r2,8)$\}$

Note that H_1 and H_2 correspond to two words in which the triggers for both scenarios are not executed and hence both s_1 and s_2 are vacuously satisfied. Augmenting the same program with the constraints:

$\leftarrow$ 0 [lpoc(t1,P_1,P_2,R):position(P_1):position(P_2):run(R)] 0.
$\leftarrow$ 0 [lpoc(t2,P_1,P_2,R):position(P_1):position(P_2):run(R)] 0.

results in no solutions in which both triggers may be non-vacuously satisfied. The reason no solution can be found is because s_1 and s_2 are inconsistent in a specification that forces the triggers to be executed. The scenario s_1 requires either *scan* or *send_instructions* to occur after the sequence *request_change*, *approve*, *send_signal* and before any other action in the scope can occur, whereas s_2 requires *update_details* to be executed after the same sequence and before any other action in the scope can occur. Since {*update_details*, *scan*, *send_instructions*} are in the scope of both scenarios, none of them may be executed, and hence the main charts for both scenarios may not be satisfied, violating constraints (EC10) and (EC11).

Single-state Fluent Properties

Single-state fluent properties are assertions that are required to hold in every state of a system. They are well-formed Boolean expressions ϕ preceded by the "always" temporal operator, i.e., $\Box\phi$. For example, suppose we are interested in ensuring that our system will only allow behaviours where instructions are sent to aircraft only if flight detail changes have been approved. This can be expressed as $\Box\phi = \Box(\mathit{Instructions\_Sent} \rightarrow \mathit{Approved})$ where $\mathit{Instructions\_Sent} = \langle\{\mathit{send\_instructions}\}, \{\mathit{instructions\_ack}\}, \bot\rangle$. A violation to such a property is a computation tree with a node index w such that $\zeta(w)(\Box\phi)$ = false.

Given a finite set of triggered scenarios $\mathcal{S} = \mathcal{U} \cup \mathcal{E}$, an ASP solver can be used to detect violations of single-state fluent properties ($\Box\phi$) by searching for solutions to the program $\Pi = \Gamma[\mathcal{S}, \mathtt{N}] \cup \{\leftarrow \mathtt{not\ violated.}, \mathtt{violated} \leftarrow \mathtt{violated(R,P)}.\} \cup \Gamma[\Box\ \phi, \mathtt{violated}]$ where $\Gamma[\Box\phi, \theta]$ is given by the following translation which makes use of predicates θ that are uniquely introduced for each fluent subformula in ϕ:

- $\Gamma[\Box f, \theta]$ is the clause $\theta(\mathtt{R,P}) \leftarrow \mathtt{not\ holdsAt(f,R,P)}$.

- $\Gamma[\Box\neg f, \theta]$ is the clause $\theta(\mathtt{R,P}) \leftarrow \mathtt{holdsAt(f,R,P)}$.

- $\Gamma[\Box(\phi \vee \psi), \theta]$ is the clauses $\{\theta(\mathtt{R,P}) \leftarrow \theta^i(\mathtt{R,P}), \theta^j(\mathtt{R,P}).\} \cup \Gamma[\Box\phi, \theta^i] \cup \Gamma[\Box\psi, \theta^j]$

- $\Gamma[\Box(\phi \wedge \psi), \theta]$ is the clauses $\{\theta(\mathtt{R,P}) \leftarrow \theta^i(\mathtt{R,P}).\} \cup \{\theta(\mathtt{R,P}) \leftarrow \theta^j(\mathtt{R,P}).\} \cup \Gamma[\Box\phi, \theta^i] \cup \Gamma[\Box\psi, \theta^j]$.

- $\Gamma[\Box(\phi \rightarrow \psi), \theta]$ is the clauses $\{\theta(\mathtt{R,P}) \leftarrow \theta^i(\mathtt{R,P}),\ \theta^j(\mathtt{R,P}).\} \cup \Gamma[\Box\neg\phi, \theta^i] \cup \Gamma[\Box\psi, \theta^j]$.

A solution Δ to the above program represents a computation tree where the formula ϕ is violated by at least one word w_H defined by $H \subset \Delta$ where $\zeta(w_H)(\phi)$ = false. For instance the property $\Box(\mathit{Instructions\_Sent} \rightarrow \mathit{Approved})$ is represented as:

```
q1(R,P)← holdsAt(instructions_sent,R,P).
q2(R,P)← not holdsAt(approved,R,P).
violated(R,P) ← q1(R,P),q2(R,P).
```

Running the ASP solver on Π results in a number of solutions including one which contains the atoms `violated(r2,8)`, `violated(r2,7)` and `violated(r1,8)` and the runs:

H_1 = {`happens(request_change,r1,0) happens(approve,r1,1)`
`happens(send_details,r1,2) happens(send_signal,r1,3)`

happens(scan,r1,4) happens(instructions_ack,r1,5)
happens(send_instructions,r1,6) happens(update_details,r1,7)
happens(update_details,r1,8) }
H_2 = { happens(request_change,r2,0) happens(approve,r2,1)
happens(send_details,r2,2) happens(send_signal,r2,3)
happens(send_instructions,r2,4) happens(scan,r2,5)
happens(update_details,r2,6) happens(send_signal,r2,7)
happens(change_ack,r2,8) }

6 Discussion and Related Work

As discussed in Section 2, existing TSs analysis algorithms and tools [?] perform analysis of a collection of TSs using a *modal transition system* (MTS) [?] that is generated via an incomplete translation. The generated MTS is a transition system that represents a collection of possible computation trees that satisfy the given TSs. Returning to our example, the algorithm in [?] results in an MTS that includes computation trees in which *update_details* can only be executed before *send_instructions*. This is because it produces an MTS which represents a collection of computation trees where the action *update_details* must occur after the execution of *send_instructions* action. Although further extensions of MTSs have recently been developed (e.g., DMTS [?]), these are still not complete in the general case. Our approach overcomes these difficulties.

The modularity of our EC representation allows us to easily adjust our translation to accommodate various alternative interpretations that have been suggested for TSs. In particular, our translation distinguishes between uTSs and eTSs only by the constraint (EC11). In [?] a weaker interpretation of a uTS is proposed that does not require all linearisations of the main chart to be exhibited by the system. This can be accommodated for in our translation simply by a weaking of (EC10) with the condition `not uni(S)`. This weaker interpretation (and other variations proposed) cannot be captured using MTSs.

The EC representation used in this paper supports partial specifications that use both existential and universal narratives under linear and branching time. Thus, not only can it support reasoning about languages such as Message Sequence Charts, Sequence Diagrams and LSCs, but also extensions of them, including extensions that support use-case-like constructs that are existential and branching in nature.

Our representation of the basic EC axioms in ASP follows the spirit of the work presented in [?, ?] but differs in that we introduce a notion of parallel runs. Bounded verification in ASP has been explored previously in [?, ?]. Our work differs in many ways, not least in that we present a technique for translating TSs which have branching time semantics into ASP, whereas the work done in [?, ?] is concerned with specifications of linear semantics. Our solutions represent computation trees rather than individual runs. [?] describes an approach for representing fluent linear temporal logic (FLTL) expressions as EC logic programs and demonstrates how Inductive Logic Programming can be used to refine MTSs that satisfy the FLTL expressions from a given set of runs representing ‘good’ and ‘bad’ system executions. Our work is however concerned with formalising and reasoning about triggered scenarios, whose semantics differ from those of

FLTL expressions, in ASP, and using ASP to generate examples of computation trees that violate some given property.

7 Conclusions and Future Work

The aim of this paper is to lay a foundation for the logic-based analysis of TSs. To achieve this we have introduced extensions to the EC to model notions of multiple time-lines and same action histories. We have then defined a general translation of TSs into this framework into finite time and demonstrated how the ASP representation may be used as an alternative verification method for detecting vacuity and violation of single-state fluent properties. The soundness and utility of our translation in comparison to existing synthesis algorithms has been discussed. Our future work will focus on developing a comprehensive ASP-based analysis framework for verifying the correctness of TS specifications with respect to general forms of safety properties as well as liveness properties. We will also investigate the use of inductive learning methods for revising our ASP representation of TSs to eliminate existing violations.

Acknowledgements: Many thanks to German Sibay for insightful discussions. This work is partly financed by ERC project PBM - FIMBSE (No. 204853).

A Translation of Triggered Scenarios into ASP

In this appendix we provide the translation of the triggered scenario given in Fig. 1.A into an EC-based answer set programming representation. Full details of programs used in this paper are available to download at (*www.doc.ic.ac.uk/~da04/CaseStudies*).

```
trigger_scenario(uTS_SendInstructionsAndScan).
uni(uTS_SendInstructionsAndScan).
trigger(t1).
is_in(t1, uTS_SendInstructionsAndScan).
main(m1).
is_in(m1, uTS_SendInstructionsAndScan).
scope(approve, uTS_SendInstructionsAndScan).
scope(send_signal, uTS_SendInstructionsAndScan).
scope(request_change, uTS_SendInstructionsAndScan).
scope(send_instructions, uTS_SendInstructionsAndScan).
scope(scan, uTS_SendInstructionsAndScan).
```

```
lpoc(t1, $P_{start}$, $P_{end}$, R) ←
    run(R), position($P_{start}$), position($P_{end}$),
    $P_{start} < P_{end}$,
    position($P_1$), position($P_2$), position($P_3$),
    $P_{start} \leq P_1$, $P_1 < P_2$, $P_2 < P_3$, $P_3 < P_{end}$,
    happens(request_change, R, $P_1$),
    happens(approve, R, $P_2$),
    happens(send_signal, R, $P_3$),
    not scoped_action_happens(t1, $P_{start}$, $P_{end}$, R, $P_1$, $P_2$, $P_3$).

scoped_action_happens(t1, $P_{start}$, $P_{end}$, R, $P_1$, $P_2$, $P_3$) ←
    position($P_{start}$), position($P_{end}$), run(R),
    position($P_1$), position($P_2$), position($P_3$),
    is_in(t1, uTS_SendInstructionsAndScan),
    action(A), scope(A, uTS_SendInstructionsAndScan),
    position(P), $P_{start} \leq P$, $P < P_{end}$, $P \neq P_1$, $P \neq P_2$, $P \neq P_3$,
    happens(A, R, P).

lpoc(m1, $P_{start}$, $P_{end}$, R) ←
    position($P_{start}$), position($P_{end}$), run(R),
    $P_{start} < P_{end}$,
    position($P_1$), position($P_2$),
    $P_{start} \leq P_1$, $P_1 < P_{end}$, happens(send_instructions, R, $P_1$),
    $P_{start} \leq P_2$, $P_2 < P_{end}$, happens(scan, R, $P_2$),
    not scoped_action_happens(m1, $P_{start}$, $P_{end}$, R, $P_1$, $P_2$).

scoped_action_happens(m1, $P_{start}$, $P_{end}$, R, $P_1$, $P_2$) $\leftarrow$
    position($P_{start}$), position($P_{end}$), run(R),
    position($P_1$), position($P_2$),
    is_in(m1, uTS_SendInstructionsAndScan),
    action(A), scope(A, uTS_SendInstructionsAndScan),
    position(P), $P_{start} \leq$ P, P< $P_{end}$, P$\neq$ $P_1$, P$\neq$ $P_2$,
    happens(A, R, P).

main_chart(SM, P, R) $\leftarrow$
    position(P), run(R),
    position($P_1$), P < $P_1$,
    lpoc(SM, P, $P_1$, R).

linearisation_id(1).

linearisation(m1, 1, P, R) $\leftarrow$
    position(P), run(R),
    position($P_1$), position($P_2$), P$\leq$ $P_1$, P$\leq$ $P_2$, $P_1$ < $P_2$,
    happens(send_instructions, R, $P_1$),
    happens(scan, R, $P_2$),
    not scoped_action_happens(m1, P, $P_2$, R, $P_1$, $P_2$).

linearisation_id(2).

linearisation(m1, 2, P, R) $\leftarrow$
    position(P), run(R),
    position($P_1$), position($P_2$), P$\leq$ $P_1$, P$\leq$ $P_2$, $P_1$ < $P_2$,
    happens(scan, R, $P_1$),
    happens(send_instructions, R, $P_2$),
    not scoped_action_happens(m1, P, $P_2$, R, $P_1$, $P_2$).

execution_state($e_2$).
execution_state($e_{23}$).
execution_state($e_{234}$).
execution_state($e_{2347}$).
execution_state($e_{2348}$).
trigger_complete($e_{234}$).
trigger_complete($e_{2347}$).
trigger_complete($e_{2348}$).
```

```
holds_execution_state(e2, R, P) ←
     run(R), position(P),
     position(P_start), position(P1),
      P_start ≤ P1, P1 ≤ P,
     happens(request_change, R, P1),
     not scoped_action_happens(t1, P_start, P, R, P1).

scoped_action_happens(t1, P_start, P_end, R, P1) ←
     position(P_start), position(P_end), run(R),
     position(P1),
     is_in(t1, uTS_SendInstructionsAndScan),
     action(A), scope(A, uTS_SendInstructionsAndScan),
     position(P), P_start  ≤ P, P< P_end, P≠ P1,
     happens(A, R, P).

holds_execution_state(e23, R, P) ←
     run(R), position(P),
     position(P_start), position(P1), position(P2),
     P_start ≤ P1,P1 ≤ P2, P2 ≤ P,
     happens(request_change, R, P1),
     happens(approve, R, P2),
     not scoped_action_happens(t1, P_start, P, R, P1, P2).

scoped_action_happens(t1, P_start, P_end, R, P1, P2) ←
     position(P_start), position(P_end), run(R),
     position(P1), position(P2),
      is_in(t1, uTS_SendInstructionsAndScan),
     action(A), scope(A, uTS_SendInstructionsAndScan),
     position(P), P_start  ≤ P, P< P_end, P≠ P1, P≠ P2,
     happens(A, R, P).

holds_execution_state(e234, R, P) ←
     run(R), position(P),
     position(P_start), position(P1), position(P2), position(P3),
      P_start ≤ P1, P1 ≤ P2, P2 < P3, P3 ≤ P,
     happens(request_change, R, P1),
     happens(approve, R, P2),
     happens(send_signal, R, P3),
     not scoped_action_happens(t1, P_start, P, R, P1, P2, P3).

holds_execution_state(e2347, R, P) ←
      run(R), position(P), position(P_start), position(P1), position(P2)
      position(P3), position(P4),
     $P_{start} \leq P_1, P_1 \leq P_2, P_2 < P_3, P_3 < P_4, P_4 \leq P$,
      happens(request_change, R, $P_1$), happens(approve, R, $P_2$),
      happens(send_signal, R, $P_3$), happens(send_instructions, R, $P_4$)
      not scoped_action_happens(m1, $P_{start}$, P, R, $P_1$, $P_2$, $P_3$, $P_4$).

scoped_action_happens(m1, $P_{start}$, $P_{end}$, R, $P_1$, $P_2$, $P_3$, $P_4$) $\leftarrow$
     position($P_{start}$), position($P_{end}$), run(R),
     position($P_1$), position($P_2$), position($P_3$), position($P_4$),
      is_in(m1, uTS_SendInstructionsAndScan),
     action(A), scope(A, uTS_SendInstructionsAndScan),
     position(P), $P_{start} \leq P$, $P < P_{end}$, $P \neq P_1$, $P \neq P_2$, $P \neq P_3$, $P \neq P_4$,
     happens(A, R, P).

holds_execution_state($e_{2347}$, R, P) $\leftarrow$
     run(R), position(P),
     position($P_{start}$), position($P_1$), position($P_2$),
     position($P_3$), position($P_4$),
     $P_{start} \leq P_1, P_1 \leq P_2, P_2 < P_3, P_3 < P_4, P_4 \leq P$,
     happens(request_change, R, $P_1$),
     happens(approve, R, $P_2$),
     happens(send_signal, R, $P_3$),
     happens(scan, R, $P_4$)
     not scoped_action_happens(m1, $P_{start}$, P, R, $P_1$, $P_2$, $P_3$, $P_4$).
```

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