Reasoning on LTL on Finite Traces: Insensitivity to Infiniteness

Giuseppe De Giacomo
Riccardo De Masellis
Dip. di Ing. Informatica, Automatica e Gestionale
Sapienza Università di Roma, Italy
degiacomo@di.uniroma1.it
demasellis@di.uniroma1.it

Marco Montali
KRDB Research Centre for Knowledge and Data
Free University of Bozen-Bolzano, Italy
montali@inf.unibz.it

Abstract
In this paper we study when an LTL formula on finite traces (LTL$_f$) is insensitive to infiniteness, that is, it can be correctly handled as a formula on infinite traces under the assumption that at a certain point the infinite trace starts repeating an end event forever, trivializing all other propositions to false. This intuition has been put forward and (wrongly) assumed to hold in general in the literature. We define a necessary and sufficient condition to characterize whether an LTL$_f$ formula is insensitive to infiniteness, which can be automatically checked by any LTL reasoner. Then, we show that typical LTL$_f$ specification patterns used in process and service modeling in CS, as well as trajectory constraints in Planning and transition-based LTL$_f$ specifications of action domains in KR, are indeed very often insensitive to infiniteness. This may help to explain why the assumption of interpreting LTL on finite and on infinite traces has been (wrongly) blurred. Possibly because of this blurring, virtually all literature detours to Büchi automata for constructing the NFA that accepts the traces satisfying an LTL$_f$ formula. As a further contribution, we give a simple direct algorithm for computing such NFA.

1 Introduction
LTL on finite traces, here called LTL$_f$, as in (De Giacomo and Vardi 2013), has been extensively used in AI. For example, it is at the base of trajectory constraints for Planning in PDDL 3.0 (Bacchus and Kabanza 2000; Gerevini et al. 2009), the de-facto standard formalism for representing planning problems. Notably, LTL$_f$ is recently gaining momentum in CS as a declarative way to specify (terminating) services and processes (Pesic and van der Aalst 2006; Montali et al. 2010; Sun, Xu, and Su 2012). We will collectively refer here to this literature as the DECLARE approach, after the main system in that area (Pesic, Schonenberg, and van der Aalst 2007).

The presence of a big body of work in LTL on infinite traces (Gabbay et al. 1980; Vardi 1996; Holzmann 1995), leads researchers to “hack” it for dealing with finite traces as well, “blurring” the distinction between the two settings. For example, both the declarative patterns for processes and services, widely adopted in the DECLARE approach (Pesic and van der Aalst 2006; Montali et al. 2010) are directly inspired by a catalogue of temporal logic patterns developed for LTL on infinite traces (Dwyer, Avrunin, and Corbett 1999), as the trajectory constraints in PDDL 3.0 are. As another example, in (Edelkamp 2006) it is proposed to directly use Büchi automata, capturing LTL on infinite traces, for LTL$_f$, saying: “[...] we can cast the Büchi automaton as an NFA (nondeterministic finite automaton, ed.), which accepts a word (i.e., trace ed.) if it terminates in a final state.” Then in (Gerevini et al. 2009) this is taken up, saying: “Since PDDL 3.0 constraints are normally evaluated over finite trajectories, the Büchi acceptance condition, that “an accepting state is visited infinitely often”, reduces to the standard acceptance condition that the automaton is in an accepting state at the end of the trajectory.” (Notice: this is incorrect if one simply leaves as accepting states those of the Büchi automaton.)

In (van der Aalst and Pesic 2006) the authors gave a quite appealing, but unfortunately incorrect in general, intuition for the blurring: “[...] we use the original algorithm for the generation of ( Büchi, ed.) automata, but we slightly change the DecSerFlow (i.e., DECLARE, ed.) model before creating the automaton. To be able to check if a finite trace is accepted, we add one “invisible” activity and one “invisible” constraint to every DecSerFlow model and then construct the automaton. With this we specify that each execution of the model will eventually end. We introduce an “invisible” activity $e$, which represents the ending activity in the model. We use this activity to specify that the service will end - the termination constraint. This constraint has the LTL formula $\Diamond e \land \Box (e \rightarrow O e)$.” In DECLARE it is assumed that only one activity can happen (i.e., only a proposition is true) at every time point, so the presence of the “$e(nd)$” activity above implies that all other propositions trivialize to false.

In fact, the two variants of LTL on finite and infinite traces are quite different, as discussed, e.g., in (Baier and McIraith 2006; De Giacomo and Vardi 2013). So, why can the research community live up with this blurring between finite and infinite traces? We help to answer this question in this paper by showing that the intuition in (van der Aalst and Pesic 2006) reported above is surprisingly correct over several widely used formulas. Specifically, we define the notion of insensitivity to infiniteness for an LTL$_f$ formula, which captures exactly the intuition in (van der Aalst and Pesic 2006).\footnote{Note that in (Bauer and Haslum 2010) a similar idea is considered, for finite traces that are extended by repeating at infinitum the...}
We, then, define a necessary and sufficient condition, which can be automatically checked by any LTL reasoner, to verify whether an LTL$_f$ formula is insensitive to infiniteness. Using such a condition, we show that all LTL$_f$ formulas corresponding to the DECLARE patterns but one, are indeed insensitive to infiniteness. We also show that virtually all transition-based specifications of action domains expressed in LTL$_f$ are insensitive to infiniteness, and that most PDDL 3.0 trajectory constraints can be easily adjusted to meet this property. Possibly because of the blurring between finite and infinite traces, virtually all literature in AI and CS detours to Büchi automata for building the NFA that accepts the traces satisfying an LTL$_f$ formula (Giannakopoulou and Havelund 2001; Edelkamp 2006; Baier and McIlraith 2006; Baier, Katoen, and Guildsland Larsen 2008; Bauer and Haslum 2010; Westergaard 2011). As a further contribution, we give a simple direct algorithm for computing such NFA.

2 LTL$_f$: LTL on Finite Traces

LTL$_f$ (De Giacomo and Vardi 2013) uses the same syntax of the original LTL (Pnueli 1977). Formulas of LTL$_f$ are built from a set $P$ of propositional symbols and are closed under the boolean connectives, the unary temporal operator $\square$ (next-time) and the binary temporal operator $\mathcal{U}$ (until):

$$\varphi ::= a \mid \neg \varphi \mid \varphi_1 \land \varphi_2 \mid \square \varphi \mid \varphi_1 \mathcal{U} \varphi_2 \quad \text{with } a \in P$$

Intuitively, $\square \varphi$ says that $\varphi$ holds at the next instant, $\varphi_1 \mathcal{U} \varphi_2$ says that at some future instant $\varphi_2$ will hold and until that point $\varphi_1$ holds. Common abbreviations are also used, including the ones listed below.

- Standard boolean abbreviations, such as true, false, $\lor$, $\land$.
- last $= \neg \square true$ denotes the last instant of the sequence. Over infinite traces it corresponds to $\neg false$ and is indeed always false, while in LTL$_f$ it becomes true at the last instance of the sequence.

- $\neg \square \varphi = \neg \neg \varphi$ is interpreted as a weak next, stating that if last does not hold then $\varphi$ must hold in the next state.

- $\square \varphi \equiv true \mathcal{U} \varphi$ says that $\varphi$ will eventually hold before the last instant (included).

- $\square \varphi \equiv \neg \square \neg \varphi$ says that from the current instant till the last instant $\varphi$ will always hold.

- $\varphi_1 \mathcal{U} \varphi_2 = \neg (\varphi_1 \mathcal{U} \neg \varphi_2)$ means that $\varphi_1$ releases $\varphi_2$, i.e., either $\varphi_2$ must hold forever, or until $\varphi_1$ also holds.

- $\varphi_1 \mathcal{W} \varphi_2 = (\varphi_1 \mathcal{U} \varphi_2 \lor \Box \varphi_1)$ is interpreted as a weak until, and means that $\varphi_1$ holds until $\varphi_2$ or forever.

The semantics of LTL$_f$ is given in terms of finite traces denoting a finite sequence of consecutive instants of time, i.e., finite words $w$ over the alphabet of $2^P$, containing all possible interpretations of the propositional symbols in $P$. Given a finite trace $w$, we inductively define when an LTL$_f$ formula $\varphi$ is true at an instant $i$ (for $0 \leq i \leq n$), written $\pi, i \models \varphi$, as:

- $\pi, i \models a$, for $a \in P$ iff $a \in \pi(i)$.
- $\pi, i \models \neg \varphi$ iff $\pi, i \not\models \varphi$.
- $\pi, i \models \varphi_1 \land \varphi_2$ iff $\pi, i \models \varphi_1$ and $\pi, i \models \varphi_2$.
- $\pi, i \models \Box \varphi$ iff $i < n$ and $\pi, i+1 \models \varphi$.

Theorem 1. (De Giacomo and Vardi 2013) Satisfiability (hence validity and logical implication) for LTL$_f$ formulas is PSPACE-complete.

We observe that LTL on infinite traces and LTL$_f$ are quite different. E.g., the formula

$$\diamond a \land \diamond (b \rightarrow \diamond b) \land \diamond (\neg a \lor \neg b) \quad (1)$$

is unsatisfiable in LTL$_f$ but is satisfiable in LTL. In other words, in a finite trace setting $\diamond a \land \diamond (b \rightarrow \diamond b) \land \diamond (\neg a \lor \neg b)$ implies that eventually both $a$ and $b$ are going to be simultaneously true. Interestingly, the NFA for (1) on finite traces and the Büchi automaton for the same formula on infinite traces are radically different. In fact, the NFA recognizes nothing (cf. Figure 1b), while the Büchi automaton is shown in Figure 1a. Certainly, one cannot consider such Büchi automaton as a correct NFA for the formula on finite traces by simply considering the accepting states as final.

3 Insensitivity to Infiniteness

Often the distinction between interpreting LTL formulas over finite vs. infinite traces is blurred via some hacking. In this section we want to tackle this issue in a precise way.

One can reduce LTL$_f$ into LTL (on infinite traces), while preserving all standard reasoning task, such as satisfiability, validity, etc. In particular, given an LTL$_f$ formula $\varphi$ we can construct a corresponding LTL formula, as follows: (i) introduce a fresh proposition “end” to denote that the trace is ended (note that end $\not\in P$, and that last is true just before the first occurrence of end); (ii) require that end eventually holds ($\diamond end$); (iii) require that once end becomes true it stays true forever ($\diamond (end \rightarrow \diamond end)$); (iv) require that when end is true, all other propositions are reset to false ($\square(\diamond \neg a)$).

Figure 1: Automata for formula (1)

- $\pi, i \models \varphi_1 \mathcal{U} \varphi_2$ iff for some $j$ s.t. $i \leq j \leq n$, we have $\pi, j \models \varphi_2$, and for all $k, i < k < j$, we have $\pi, k \models \varphi_1$.

A formula $\varphi$ is true in $\pi$, in notation $\pi \models \varphi$, if $\pi, 0 \models \varphi$. A formula $\varphi$ is satisfiable if it is true in some finite trace, and it is valid if it is true in every finite trace. A formula $\varphi$ logically implies a formula $\varphi'$, written $\varphi \models \varphi'$, if for every finite trace $\pi$ we have that $\pi \models \varphi$ implies $\pi \models \varphi'$. Notice that satisfiability, validity and logical implication are all mutually reducible to each other: for example $\varphi$ is valid iff $\neg \varphi$ is unsatisfiable. Similarly, $\varphi \models \varphi'$ iff $\varphi \land \neg \varphi'$ is unsatisfiable.

Given a formula on finite traces, we translate the LTL$_f$ formula into an LTL formula as follows:

$$f(a) \rightarrow a \quad f(\varphi_1 \lor \varphi_2) \rightarrow f(\varphi_1) \lor f(\varphi_2) \quad f(\neg \varphi) \rightarrow \neg f(\varphi) \quad f(\circ \varphi) \rightarrow \circ (f(\varphi) \land \neg end) \quad f(\mathcal{U} \varphi) \rightarrow (f(\varphi) \lor \neg end)$$

Theorem 2. Let $\pi_1$ be an infinite trace. Then

$$\pi_1 \models \diamond \neg a \land \square (\neg a) \land \neg a \quad (a \in P)$$

iff it is has the form $\pi_1 = \pi_f \{a\}^\omega$, where $\pi_f$ is always false in $\pi_f$. 

propositional assignment in the last element of the finite trace. Their results can be considered complementary to ours.
Proof (sketch). The only if direction is immediate. For the if direction, suffice it to observe that if \( \pi_i \) satisfies \( \varphi \land \Box (\text{end} \rightarrow \Box \text{end}) \land \Box (\text{end} \rightarrow \bigwedge_{a \in P} \lnot a) \) then there must be a first instant in which end becomes true, hence \( \pi_i \) must have the form \( \pi_f \{ \text{end} \} \omega \) where end is always false in \( \pi_f \).

**Theorem 3.** Let \( \varphi \) be an LTL\(_f\) formula and \( \pi_i = \pi_f \{ \text{end} \} \omega \) an infinite trace where \( \text{end} \) is always false in \( \pi_f \). Then

\[
\pi_f \models \varphi \iff \pi_f \{ \text{end} \} \omega \models f(\varphi).
\]

Proof (sketch). Both direction can be shown by induction on the structure of the formula \( \varphi \).

We now exploit the formal notions behind the above two theorems to define the notion of insensitivity to infiniteness, capturing the intuition discussed in the introduction.

**Definition 1.** An LTL\(_f\) formula \( \varphi \) is insensitive to infiniteness if for every (infinite) trace \( \pi_i = \pi_f \{ \text{end} \} \omega \) where \( \text{end} \) is always false in \( \pi_f \), we have that

\[
\pi_f \models \varphi \iff \pi_f \{ \text{end} \} \omega \models \varphi.
\]

LTL\(_f\) formulas that are insensitive to infiniteness can be translated into LTL by simply adding the conditions on end without applying the translation function \( f(\cdot) \). Notice that if an LTL\(_f\) formula is insensitive to infiniteness, we can essentially blur the distinction between finite and infinite traces by simply asserting in the infinite case that there exists an end of the significant part and that once such end is reached every proposition is trivially reset to false in the infinite trace.

Next theorem gives us necessary and sufficient conditions for an LTL\(_f\) formula to be insensitive to infiniteness.

**Theorem 4.** An LTL\(_f\) \( \varphi \) is insensitive to infiniteness if and only if the following LTL formula is valid:

\[
(\Diamond \text{end} \land \Box (\text{end} \rightarrow \Box \text{end}) \land \Box (\text{end} \rightarrow \bigwedge_{a \in P} \lnot a)) \rightarrow (\varphi \equiv f(\varphi)).
\]

Proof. (If direction.) By Theorem 2, we know that for every infinite trace satisfying the premise of the implication must have the form \( \pi_i = \pi_f \{ \text{end} \} \omega \) where end is always false in \( \pi_f \). While by Theorem 3 \( \pi_f \models \varphi \) if and only if \( \pi_f \{ \text{end} \} \omega \models f(\varphi) \). But then by the consequent of the implication we have that \( \pi_f \{ \text{end} \} \omega \models \varphi \), hence \( \varphi \) is insensitive to infiniteness.

(Only if direction.) Since \( \varphi \) is insensitive to infiniteness, we have that for every (infinite) trace \( \pi_i = \pi_f \{ \text{end} \} \omega \) where end is always false in \( \pi_f \); \( \pi_i \models \varphi \) if and only if \( \pi_f \{ \text{end} \} \omega \models \varphi \). On the other hand, by Theorem 3, we have that \( \pi_f \models \varphi \) if and only if \( \pi_f \{ \text{end} \} \omega \models f(\varphi) \). By combining the two above equivalences we get \( \pi_f \{ \text{end} \} \omega \models \varphi \) if and only if \( \pi_f \{ \text{end} \} \omega \models f(\varphi) \), which in turn implies \( \pi_f \{ \text{end} \} \omega \models \varphi \equiv f(\varphi) \). Now by Theorem 2 we have that an infinite trace has the form \( \pi_i = \pi_f \{ \text{end} \} \omega \) if and only if \( \pi_i \models (\Diamond \text{end} \land \Box (\text{end} \rightarrow \Box \text{end}) \land \Box (\text{end} \rightarrow \bigwedge_{a \in P} \lnot a)) \). Hence, we get \( \pi_i \models (\Diamond \text{end} \land \Box (\text{end} \rightarrow \Box \text{end}) \land \Box (\text{end} \rightarrow \bigwedge_{a \in P} \lnot a)) \) implies \( \pi_f \models \varphi \equiv f(\varphi) \), which is the claim.

This theorem is quite interesting since it gives us a technique to check an LTL\(_f\) formula for insensitivity to the infiniteness: we simply need to check the standard LTL formula

\[
(\Diamond \text{end} \land \Box (\text{end} \rightarrow \Box \text{end}) \land \Box (\text{end} \rightarrow \bigwedge_{a \in P} \lnot a)) \rightarrow (\varphi \equiv f(\varphi))
\]

for validity, or its negation for unsatisfiability, which can be done by checking for emptiness the corresponding Bichi automata, see e.g., (Vardi 1996). For example, one can check that the LTL\(_f\) formula (1) is insensitive to infiniteness.

We close the section by showing that the class of LTL\(_f\) formulas that are insensitive to infiniteness is closed under Boolean operations.

**Theorem 5.** Let \( \varphi_1 \) and \( \varphi_2 \) be two LTL\(_f\) formulas that are insensitive to infiniteness. Then the LTL\(_f\) formulas \( \lnot \varphi_i \) (\( i = 1, 2 \)) and \( \varphi_1 \land \varphi_2 \) are also insensitive to infiniteness.

Proof. By induction on the structure of the formula, considering the definition of insensitive to infiniteness.

### 4 DECLARE Process Modeling

DECLARE is a language and framework for the declarative, constraint-based modeling of processes and services. It started from seminal works on ConDec (Pesci and van der Aalst 2006) and DecSerFlow (van der Aalst and Pesci 2006; Montali et al. 2010). A thorough treatment of constraint-based processes can be found in (Pesci 2008; Montali 2010).

The DECLARE framework provides a set \( P \) of propositions representing atomic tasks (i.e., actions), which are units of work in the process. Notice that properties of states are not represented. DECLARE assumes that, at each point in time, one and only one task is executed, and that the process eventually terminates. Following the second assumption, LTL\(_f\) is used to specify DECLARE processes, whereas the first assumption is captured by the following LTL\(_f\) formula, assumed as an implicit constraint:

\[
\xi_P = \Diamond (\bigvee_{a \in P} a) \land \Diamond (\bigwedge_{a,b \in P} a \neq b \rightarrow \lnot b),
\]

which we call the DECLARE assumption.

A DECLARE model is a set \( C \) of LTL\(_f\) constraints over \( P \), used to restrict the allowed execution traces. Among all possible LTL\(_f\) constraints, some specific patterns have been singled out as particularly meaningful for expressing DECLARE processes, taking inspiration from (Dwyer, Avrunin, and Corbett 1999). As shown in Table 1, patterns are grouped into four families: (i) existence (unary) constraints, stating that the target task must/cannot be executed (a certain amount of times); (ii) choice (binary) constraints, modeling choice of execution; (iii) relation (binary) constraints, modeling that whenever the source task is executed, then the target task must also be executed (possibly with additional requirements); (iv) negation (binary) constraints, modeling that whenever the source task is executed, then the target task cannot be executed (possibly with additional restrictions).

Observe that the set of finite traces that satisfies the constraints \( C \) together with the DECLARE assumption \( \xi_P \) can be captured by a single deterministic process, obtained by:

1. generating the corresponding NFA (exponential step);
2. transforming it into a DFA- deterministic finite-state automaton (exponential step);
3. trimming the resulting DFA by removing every state from which no final state is reachable (polynomial step).

The obtained DFA is indeed a process in the sense that at every step, depending only on the history (i.e., the current state), it exposes the set of tasks that are legally executable and eventually lead to a final state (assuming fairness of the
All these services reduce to standard reasoning in LTL which tasks are currently legal, which constraints are current. We use $\pi$ with the constraints. This requires an engine that, at each

DECLARE $\pi$ the execution can be

$I$ the execution so far:

- Task $a \in \mathcal{P}$ is legal in $\pi_I$ if there exist a (finite, possibly empty) trace $\pi_2$ s.t.: $\pi_1 a \pi_2 \models C \land \xi_P$.
- Constraint $C \in \mathcal{C}$ is pending in $\pi_I$ if $\pi_1 \not\models C$.
- The execution can be ended in $\pi_I$ if $\pi_1 \models C \land \xi_P$.

All these services reduce to standard reasoning in LTL.

It turns out that all DECLARE patterns but one are insensitive to infiniteness (see Table 1).

**Theorem 6.** All the DECLARE patterns, with the exception of negation chain succession, are insensitive to infiniteness, independently from the DECLARE assumption.

The theorem can be proven automatically, making use of an

LTL reasoner on infinite traces. Specifically, each DECLARE pattern can be grounded on a concrete set of tasks (propositions), and then, by Theorem 4, we simply need to check the validity of the corresponding formula. In fact, we encoded each validity check in the model checker NuSMV$^2$, following the approach of satisfiability via model checking (D. Rozier and Vardi 2007). E.g., the following NuSMV specification checks whether response is insensitive to infiniteness:

```
MODULE main
VAR a:boolean; b:boolean; other:boolean; end:boolean;
LTLSPEC
(F(end) & G(end -> X(end)) & G(end -> (b & !a)))
-> ( ((G(a -> X(!F(b)))) <=>
    (G(a -> X(F(b & !end)))) & (!end) & !end) )
```

NuSMV confirmed that all patterns but the negation chain succession are insensitive to infiniteness. This is true both making or not the DECLARE assumption, and independently on whether $\mathcal{P}$ only contains the propositions explicitly mentioned in the pattern, or also further ones.

Let us discuss the negation chain succession, which is not insensitive to infiniteness. On infinite traces, $\Box(a \equiv \neg b)$ retains the meaning specified in Table 1. On finite traces, it also forbids $a$ to be the last-executed task in the finite trace, since it requires $a$ to be followed by another task that is different from $b$. E.g., we have that $\{a\}$ $\{end\} \models \Box(a \equiv \neg b)$, but $\{a\} \not\models \Box(a \equiv \neg b)$. This is not foreseen in the informal

### Table 1: Declare patterns and their insensitivity to infiniteness

<table>
<thead>
<tr>
<th>NAME</th>
<th>NOTATION</th>
<th>LTL$_f$ FORMALIZATION</th>
<th>DESCRIPTION</th>
<th>INSENSITIVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Existence</td>
<td>$\pi$</td>
<td>$\square a \land \Box a$</td>
<td>$a$ must be executed at least once</td>
<td>Y</td>
</tr>
<tr>
<td>Absence 2</td>
<td>$\pi$</td>
<td>$\neg \square a \land \Box a$</td>
<td>$a$ can be executed at most once</td>
<td>Y</td>
</tr>
<tr>
<td>Choice</td>
<td>$\pi$</td>
<td>$\Box a \lor \Box b$</td>
<td>$a$ or $b$ must be executed</td>
<td>Y</td>
</tr>
<tr>
<td>Exclusive Choice</td>
<td>$\pi$</td>
<td>$\square a \land \Box b \lor \square b \land \Box a$</td>
<td>Either $a$ or $b$ must be executed, but not both</td>
<td>Y</td>
</tr>
<tr>
<td>Resp. existence</td>
<td>$\pi$</td>
<td>$\Box a \land \Box b$</td>
<td>If $a$ is executed, then $b$ must be executed as well</td>
<td>Y</td>
</tr>
<tr>
<td>Coexistence</td>
<td>$\pi$</td>
<td>$\Box a \land \Box b$</td>
<td>Either $a$ and $b$ are both executed, or none of them is executed</td>
<td>Y</td>
</tr>
<tr>
<td>Response</td>
<td>$\pi$</td>
<td>$\square a \land \Box b$</td>
<td>Every time $a$ is executed, $b$ must be executed afterwards</td>
<td>Y</td>
</tr>
<tr>
<td>Precedence</td>
<td>$\pi$</td>
<td>$\Box a \land \Box b$</td>
<td>$a$ can be executed only if $b$ has been executed before</td>
<td>Y</td>
</tr>
<tr>
<td>Succession</td>
<td>$\pi$</td>
<td>$\square a \land \Box b$</td>
<td>$b$ must be executed after $a$, and $a$ must precede $b$</td>
<td>Y</td>
</tr>
<tr>
<td>Alt. Response</td>
<td>$\pi$</td>
<td>$\Box a \land \Box b$</td>
<td>Every $a$ must be followed by $b$, without any other $a$ in between</td>
<td>Y</td>
</tr>
<tr>
<td>Alt. Precedence</td>
<td>$\pi$</td>
<td>$\Box a \land \Box b$</td>
<td>Every $b$ must be preceded by $a$, without any other $b$ in between</td>
<td>Y</td>
</tr>
<tr>
<td>Alt. Succession</td>
<td>$\pi$</td>
<td>$\square a \land \Box b$</td>
<td>Combination of alternate response and alternate precedence</td>
<td>Y</td>
</tr>
<tr>
<td>Chain Response</td>
<td>$\pi$</td>
<td>$\square a \land \Box b$</td>
<td>$b$ must be executed next after $a$, and $a$ and $b$ must be executed next to each other</td>
<td>Y</td>
</tr>
<tr>
<td>Chain Precedence</td>
<td>$\pi$</td>
<td>$\Box a \land \Box b$</td>
<td>Task $b$ can be executed only immediately after $a$</td>
<td>Y</td>
</tr>
<tr>
<td>Chain Succession</td>
<td>$\pi$</td>
<td>$\Box a \land \Box b$</td>
<td>Tasks $a$ and $b$ must be executed next to each other</td>
<td>Y</td>
</tr>
<tr>
<td>Not Coexistence</td>
<td>$\pi$</td>
<td>$\Box a \land \Box b$</td>
<td>Only one among tasks $a$ and $b$ can be executed, but not both</td>
<td>Y</td>
</tr>
<tr>
<td>Neg. Succession</td>
<td>$\pi$</td>
<td>$\Box a \land \Box b$</td>
<td>Task $a$ cannot be followed by $b$, and $b$ cannot be preceded by $a$</td>
<td>Y</td>
</tr>
<tr>
<td>Neg. Chain Succession</td>
<td>$\pi$</td>
<td>$\Box a \land \Box b$</td>
<td>Tasks $a$ and $b$ cannot be executed next to each other</td>
<td>N</td>
</tr>
</tbody>
</table>

The full list of specifications is available here: http://www.inf.unibz.it/ montali/AAAI14
We often characterize an action domain by the set of allowed evolutions, each represented as a sequence of situations (Reiter 2001). To do so, we typically introduce a set of atomic facts, called fluents, whose truth value changes as the system evolves from one situation to the next because of actions. Since \( \text{LTL}_{LTL_f} \) do not provide a direct notion of action, we use propositions to denote them, as in (Calvanese, De Giacomo, and Vardi 2002). Hence, we partition \( P \) into fluents \( F \) and actions \( A \), adding structural constraint (analogous to the DECLARE assumption) such as \( \Box(\bigwedge_{a \in A} a \land \bigwedge_{b \in A, b \neq a} \neg b) \), to specify that one action must be performed to get to a new situation, and that a single action at a time can be performed. Then, the initial situation is described by a propositional formula \( \varphi_{init} \) involving only fluents, while effects can be modelled as:

\[
\Box(\varphi \rightarrow \Box(a \rightarrow \psi))
\]

where \( a \in A \), while \( \psi \) and \( \varphi \) are arbitrary propositional formulas involving only fluents. Such a formula states that performing action \( a \) under the conditions denoted by \( \varphi \) brings about the conditions denoted by \( \psi \). Alternatively, we can formalize effects through Reiter’s successor state axioms (Reiter 2001) (which also provide a solution to the frame problem), as in (Calvanese, De Giacomo, and Vardi 2002; De Giacomo and Vardi 2013), by translating the (instantiated) successor state axiom \( F(do(a, s)) \equiv \varphi^+(s) \lor (F(s) \land \neg \varphi^-) \) into the \( \text{LTL}_f \) formula:

\[
\Box(\Box a \rightarrow (\Box F \equiv \varphi^+ \lor F \land \neg \varphi^-)).
\]

In general, to specify effects we need special \( \text{LTL}_f \) formulas that talk only about the current state and the next state to capture how the domain does a transition from the current to the next state. Such formulas are called transition formula, and are inductively built as follows:

\[
\varphi ::= \phi \mid \Box \phi \mid \neg \varphi \mid \varphi_1 \land \varphi_2, \quad \text{where } \phi \text{ is propositional.}
\]

For such formulas we can state a notable result: under the assumption that at every step at least one proposition is true, every specification based on transition formulas is insensitive to infiniteness. More precisely:

\[
\text{A formula like } \Box((\varphi \rightarrow \Box(a \rightarrow \varphi)) \text{ corresponds to a frame axiom expressing that } \varphi \text{ does not change when performing } a.
\]
As explained in (Westergaard 2011), the DECLARE environment uses the automaton construction in (Giannakopoulou and Havelund 2001), which applies the traditional Büchi automaton construction in (Gerth et al. 1995), and then suitably defines which states have to be considered as final. The language, however, does not include the next operator. Inspired by (Giannakopoulou and Havelund 2001), also the approach in (Baier and McIlraith 2006) relies on the procedure in (Gerth et al. 1995) to build the NFA, but it implements the full LTL semantics by dealing also with the next operator.

Here, we provide a simple direct algorithm for computing the NFA corresponding to an LTL formula. The correctness of the algorithm is based on the fact that (i) we can associate with each LTL formula \( \varphi \) a polynomial alternating automaton on words \( (\text{AFW}) \) \( A_\varphi \) that accept exactly the traces that make \( \varphi \) true (De Giacomo and Vardi 2013), and (ii) every AFW can be transformed into an NFA, see, e.g., (De Giacomo and Vardi 2013).

To formulate the algorithm we do not need these notions, but we can work directly on the LTL formula. We assume our formula to be in negation normal form, by exploiting abbreviations and pushing negation inside as much as possible, leaving it only in front of propositions (any LTL formula can be transformed into negation normal form in linear time). We also assume \( P \) to include a special proposition \( \text{last} \) which denotes the last element of the trace. Note that \( \text{last} \) can be defined as \( \text{last} \equiv \top \). Then we define an auxiliary function \( \delta \) that takes an LTL formula \( \psi \) (in negation normal form) and a propositional interpretation \( \Pi \) for \( P \) (including \text{last} term), returning a positive boolean formula whose atoms are (quoted) \( \psi \) subformulas.

\[
\begin{align*}
\delta(\text{true}, \Pi) &= \text{true} \quad \text{if } a \in \Pi \\
\delta(\text{false}, \Pi) &= \text{false} \quad \text{if } a \notin \Pi \\
\delta(\neg a, \Pi) &= \text{true} \quad \text{if } a \notin \Pi \\
\delta(\varphi_1 \land \varphi_2, \Pi) &= \delta(\varphi_1, \Pi) \land \delta(\varphi_2, \Pi) \\
\delta(\varphi_1 \lor \varphi_2, \Pi) &= \delta(\varphi_1, \Pi) \lor \delta(\varphi_2, \Pi) \\
\delta(a = O \varphi, \Pi) &= \begin{cases} \varphi \quad \text{if } \text{last} \notin \Pi \\
\text{false} \quad \text{if } \text{last} \in \Pi 
\end{cases} \\
\delta(a = U \varphi, \Pi) &= \delta(a = U \varphi_2, \Pi) \lor \delta(\varphi_1, \Pi) \\
\delta(\varphi_1 = R \varphi_2, \Pi) &= \delta(\varphi_2, \Pi) \lor \delta(\varphi_1, \Pi) \\
\delta(\Diamond \varphi, \Pi) &= \text{true} \quad \text{if } \text{last} \in \Pi \\
\delta(\Box \varphi, \Pi) &= \delta(\varphi, \Pi) \land \delta(\Box \varphi, \Pi) \\
\delta(\varphi_1 \equiv \varphi_2, \Pi) &= \delta(\varphi_2, \Pi) \lor \delta(\varphi_1, \Pi) \\
\delta(\varphi_1 \implies \varphi_2, \Pi) &= \delta(\varphi_1, \Pi) \land \delta(\varphi_2, \Pi) \\
\delta(\varphi_1 \implies \neg \varphi_2, \Pi) &= \delta(\varphi_2, \Pi) \lor \neg \delta(\varphi_1, \Pi) \\
\delta(\varphi_1 \equiv \neg \varphi_2, \Pi) &= \neg \delta(\varphi_1, \Pi) \land \delta(\varphi_2, \Pi) \\
\delta(\varphi_1 \implies \varphi_2, \Pi) &= \delta(\varphi_2, \Pi) \lor \neg \delta(\varphi_1, \Pi) \\
\delta(\varphi_1 \equiv \neg \varphi_2, \Pi) &= \neg \delta(\varphi_1, \Pi) \land \delta(\varphi_2, \Pi) \\
\delta(\varphi_1 \implies \neg \varphi_2, \Pi) &= \neg \delta(\varphi_1, \Pi) \land \delta(\varphi_2, \Pi) \\
\delta(\varphi_1 \equiv \neg \varphi_2, \Pi) &= \neg \delta(\varphi_1, \Pi) \land \delta(\varphi_2, \Pi)
\end{align*}
\]

Using function \( \delta \) we can build the NFA \( A_\varphi \) of an LTL formula \( \varphi \) in a forward fashion. States of \( A_\varphi \) are sets of atomic (recall that each atom is quoted \( \psi \) subformulas) to be interpreted as a conjunction; the empty conjunction \( \emptyset \) stands for \text{true}.

1: \textbf{algorithm} LTL\_2NFA() \\
2: \textbf{input} LTL formula \( \varphi \) \\
3: \textbf{output} NFA \( A_\varphi = (\mathcal{P}, \mathcal{S}, \{s_0\}, \varphi, \{s_f\}) \) \\
4: \textbf{local} \( s_0 \leftarrow \{\varphi\} \quad \triangleright \text{single initial state} \\
5: s_f \leftarrow \emptyset \quad \triangleright \text{single final state} \\
6: \mathcal{S} \leftarrow \{s_0, s_f\}, q \leftarrow \emptyset \\
7: \textbf{while} (\mathcal{S} \text{ or } q \text{ change}) \textbf{do} \\
8: \quad \text{if} (q \in \mathcal{S} \text{ and } q' = \bigwedge (\psi \in q) \delta(\psi^n, \Pi)) \textbf{then} \\
9: \quad \quad \mathcal{S} \leftarrow \mathcal{S} \cup \{q'\} \quad \triangleright \text{update set of states}
References


De Giacomo, G., and Vardi, M. Y. 2013. Linear temporal logic and linear dynamic logic on finite traces. In *Proc. of the 23rd Int. Joint Conf. on Artificial Intelligence (IJCAI)*. IJCAI/AAAI.


Edelkamp, S. 2006. On the compilation of plan constraints and preferences. In *Proc. of the 16th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 374–377. AAAI.


1033