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,demasellis}@dis.uniroma1.it

Abstract

In this paper we study when an LTL formula on finite traces ($\text{LTL}_f$ formula) 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 $\text{LTL}_f$ formula is insensitive to infiniteness, which can be automatically checked by any LTL reasoner. Then, we show that typical $\text{LTL}_f$ specification patterns used in process and service modeling in CS, as well as trajectory constraints in Planning and transition-based $\text{LTL}_f$ 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 $\text{LTL}_f$ formula. As a further contribution, we give a simple direct algorithm for computing such NFA.

1 Introduction

LTL on finite traces, here called $\text{LTL}_f$, 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, $\text{LTL}_f$ is recently gaining momentum in CS as a declarative way to specify (terminating) services and processes (Pescic 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 (Pescic, 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 (Pescic 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 $\text{LTL}_f$, saying: “[...]

Note that in (Bauer and Haslum 2010) a similar idea is considered, for finite traces that are extended by repeating at infinitum the propositional assignment in the last element of the finite trace. Their results can be considered complementary to ours.
We, then, define a necessary and sufficient condition, which can be automatically checked by any LTL reasoner, to verify whether an LTL formula is insensitive to infiniteness. Using such a condition, we show that all LTL \( \varphi \) 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 \( \varphi \) 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 \( \varphi \) formula (Giannakopoulou and Havelund 2001; Edelkamp 2006; Baier and Mciraith 2006; Baier, Katoen, and Guldstrand Larsen 2008; Bauer and Haslum 2010; Westergaard 2011). As a further contribution, we give a simple direct algorithm for computing such NFAs.

### 2 LTL\( \varphi \): LTL on Finite Traces

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

\[
\varphi ::= a \mid \neg \varphi \mid \varphi_1 \land \varphi_2 \mid \bigcirc \varphi \mid \varphi_1 \bigcup \varphi_2 \quad \text{with} \quad a \in P
\]

Intuitively, \( \bigcirc \varphi \) says that \( \varphi \) holds at the next instant, \( \varphi_1 \bigcup \varphi_2 \) says that at some future instant \( \varphi_2 \) will hold and \( \text{until} \) that point \( \varphi_1 \) holds. Common abbreviations are also used, including the ones listed below:

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

The semantics of LTL\( \varphi \) is given in terms of finite traces denoting a finite sequence of consecutive instants of time, i.e., finite words \( \pi \) over the alphabet of \( 2^P \), containing all possible interpretations of the propositional symbols in \( P \). Given a finite trace \( \pi \), we inductively define when an LTL\( \varphi \) 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 \( a \in P \) if and only if \( a \in \pi(i) \).
- \( \pi, i \models \neg \varphi \) if \( \pi, i \not\models \varphi \).
- \( \pi, i \models \varphi_1 \land \varphi_2 \) if \( \pi, i \models \varphi_1 \) and \( \pi, i \models \varphi_2 \).
- \( \pi, i \models \bigcirc \varphi \) if \( \varphi \) is true in some finite trace, \( i < n \) and \( \pi, i+1 \models \varphi \).
- \( \pi, i \models \varphi_1 \bigcup \varphi_2 \) if for some \( j \) s.t. \( i \leq j \leq n \), we have \( \pi, j \models \varphi_2 \), and for all \( k, i \leq k < j \), 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.

We observe that LTL on infinite traces and LTL\( \varphi \) are quite different. E.g., the formula

\[
\Box a \land \Box (a \rightarrow \Box b) \land \Box (b \rightarrow \Box a) \land \Box (\neg a \lor \neg b)
\]

is unsatisfiable in LTL\( \varphi \) but is satisfiable in LTL. In other words, in a finite trace setting \( \Box a \land \Box (a \rightarrow \Box b) \land \Box (b \rightarrow \Box a) \)

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\( \varphi \) into LTL (on infinite traces), while preserving all standard reasoning tasks, such as satisfiability, validity, etc. In particular, given an LTL\( \varphi \) 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 \( \notin P \), and that last true is just before the first occurrence of end); (ii) require that end eventually holds (\( \bigcirc\text{end} \)); (iii) require that once end becomes true it stays true forever (\( \bigcirc (\text{end} \rightarrow \bigcirc\text{end}) \)); (iv) require that when end is true, all other propositions are reset to false (\( \bigcirc (\text{end} \rightarrow \bigcirc a \land \neg a) \)); (v) translate the LTL\( \varphi \) formula into an LTL formula as follows:

\[
\begin{align*}
\varphi(a) &\rightarrow a \\
\varphi(a) &\rightarrow \neg \varphi \\
\varphi &\rightarrow \bigcirc \varphi \\
\varphi &\rightarrow \bigcirc \bigcirc \varphi \\
\varphi &\rightarrow \bigcirc (\bigcirc \varphi \land \neg \varphi)
\end{align*}
\]

\[
\begin{align*}
\varphi(a) &\rightarrow a \\
\varphi(a) &\rightarrow \neg \varphi \\
\varphi &\rightarrow \bigcirc \varphi \\
\varphi &\rightarrow \bigcirc \bigcirc \varphi \\
\varphi &\rightarrow \bigcirc (\bigcirc \varphi \land \neg \varphi)
\end{align*}
\]

Theorem 2. Let \( \pi_\varphi \) be an infinite trace. Then

\[
\pi_\varphi \models \Diamond \text{end} \land \Box (\text{end} \rightarrow \Box \text{end}) \land \Box (\text{end} \rightarrow \bigcirc a) \land \Box (\text{end} \rightarrow \bigcirc a).
\]

If it is has the form \( \pi_\varphi = \pi_f \{\text{end}\}^\omega \), where end is always false in \( \pi_f \).

**Proof (sketch).** The only if direction is immediate. For the if direction, suffice it to observe that if \( \pi_f \) satisfies \( \Diamond \text{end} \land \Box (\text{end} \rightarrow \Box \text{end}) \land \Box (\text{end} \rightarrow \bigcirc a) \land \Box (\text{end} \rightarrow \bigcirc a) \)
\(\square (end \rightarrow \sigma end) \land \Box (end \rightarrow \bigwedge_{a \in P} \neg a)\) then there must be a first instant in which \(end\) becomes true, hence \(\pi_1\) must have the form \(\pi_f \{end\}^\omega\) where \(end\) is always false in \(\pi_f\).

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

**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 \(\pi_i = \pi_f \{end\}^\omega\) where \(end\) is always false in \(\pi_f\), we have that
\[
\pi_f \models \varphi \iff \pi_f \{end\}^\omega \models \varphi.
\]

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 \(f().\) 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\(_f\) formula is valid:

\[
(\diamond \diamond end \land \Box (end \rightarrow \sigma end) \land \Box (end \rightarrow \bigwedge_{a \in P} \neg a)) \rightarrow (\varphi \equiv f(\varphi))
\]

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

(Only if direction.) Since \(\varphi\) is insensitive to infiniteness, we have that for every (finite) trace \(\pi_i = \pi_f \{end\}^\omega\) where \(end\) is always false in \(\pi_f\): \(\pi_f \models \varphi\) if and only if \(\pi_f \{end\}^\omega \models \varphi\). On the other hand, by Theorem 3, we have that \(\pi_f \models \varphi\) if and only if \(\pi_f \{end\}^\omega \models f(\varphi)\). By combining the two above equivalences we get \(\pi_f \{end\}^\omega \models \varphi\) if and only if \(\pi_f \{end\}^\omega \models \varphi\), which in turn implies \(\pi_f \{end\}^\omega \models \varphi \equiv f(\varphi)\). Now by Theorem 2 we have that an infinite trace has the form \(\pi_i = \pi_f \{end\}^\omega\) if and only if \(\pi_i = (\diamond \diamond end \land \Box (end \rightarrow \sigma end) \land \Box (end \rightarrow \bigwedge_{a \in P} \neg a))\). Hence, we get \(\pi_i \models \diamond \diamond (end \rightarrow \sigma end) \land \Box (end \rightarrow \bigwedge_{a \in P} \neg a)\) implies \(\pi_i \models \varphi \equiv f(\varphi)\), 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
\[\diamond end \land \Box (end \rightarrow \sigma end) \land \Box (end \rightarrow \bigwedge_{a \in P} \neg a)\] \(
\rightarrow \varphi \equiv f(\varphi)\)
for validity, or its negation for unsatisfiability, which can be done by checking for emptiness the corresponding Büchi 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 \(\neg \varphi_1\) (\(i = 1, 2\)) and \(\varphi_1 \land \varphi_2\) are also insensitive to infiniteness.

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

## 4 DECLARE Process Modeling

DECLARE is a language and framework for the declarative, constraint-based modelling of processes and services. It started from seminal works on ConDec (Pescic and van der Aalst 2006) and DecSerFlow (van der Aalst and Pesic 2006; Montali et al. 2010). A thorough treatment of constraint-based processes can be found in (Pescic 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: \(\xi_P = \square (\bigvee_{a \in P} a) \land \bigwedge_{a,b \in P, a \neq b} a \rightarrow \neg b\), 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 execution, which disallows remaining forever in a loop). In the DECLARE implementation, this process is maintained implicit, and traces are generated incrementally, in accordance
with the constraints. This requires an engine that, at each state, infers which tasks are legal (cf. enactment below).

Three fundamental reasoning services are of interest in DECLARE: (i) Consistency, which checks whether the model is executable, i.e., there exists at least one (finite) trace \( \pi_f \) over \( \mathcal{P} \) such that \( \pi_f \models C \land \xi_P \). With a little abuse of notation, we use \( C \) to denote the LTL formula \( \bigwedge_{C \in \mathcal{C}} C \). (ii) Detection of dead tasks, which checks which model contains tasks that can never be executed; a task \( a \in \mathcal{P} \) is dead if, for every (finite) trace \( \pi_f \) over \( \mathcal{P} \): \( \pi_f \not\models (C \land \xi_P) \rightarrow \neg a. \) (iii) Enactment, which drives the execution of the model, inferring which tasks are currently legal, which constraints are currently pending (i.e., require to do something), and whether the execution can be currently ended or not; specifically, given a (finite) trace \( \pi_1 \) denoting the execution so far:

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

All these services reduce to standard reasoning in LTLf.

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 (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@!(b)) & 
    G!(a -> X@!(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 \( \{a\} \{\text{end}\}^\omega \models \Box(a \equiv \neg b) \), 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>LTLf FORMALIZATION</th>
<th>DESCRIPTION</th>
<th>INSENSITIVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Existence</td>
<td><img src="image1" alt="Existence" /></td>
<td>( \Box a )</td>
<td>( a ) must be executed at least once</td>
<td>Y</td>
</tr>
<tr>
<td>Absence 2</td>
<td><img src="image2" alt="Absence 2" /></td>
<td>( \neg (a \land \Box a) )</td>
<td>( a ) can be executed at most once</td>
<td>Y</td>
</tr>
<tr>
<td>Choice</td>
<td><img src="image3" alt="Choice" /></td>
<td>( \Box a \lor \Box b )</td>
<td>Either ( a ) or ( b ) must be executed</td>
<td>Y</td>
</tr>
<tr>
<td>Exclusive Choice</td>
<td><img src="image4" alt="Exclusive Choice" /></td>
<td>( (\Box a \lor \Box b) \land \neg (\Box a \land \Box b) )</td>
<td>Either ( a ) or ( b ) must be executed, but not both</td>
<td>Y</td>
</tr>
<tr>
<td>Resp. existence</td>
<td><img src="image5" alt="Resp. existence" /></td>
<td>( \Box a \rightarrow \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><img src="image6" alt="Coexistence" /></td>
<td>( (\Box a \rightarrow \Box b) \land (\Box b \rightarrow \Box a) )</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><img src="image7" alt="Response" /></td>
<td>( \Box(a \rightarrow \Box b) )</td>
<td>Every time ( a ) is executed, ( b ) must be executed afterwards</td>
<td>Y</td>
</tr>
<tr>
<td>Precedence</td>
<td><img src="image8" alt="Precedence" /></td>
<td>( \neg b \lor \Box a )</td>
<td>( b ) can be executed only if ( a ) has been executed before</td>
<td>Y</td>
</tr>
<tr>
<td>Succession</td>
<td><img src="image9" alt="Succession" /></td>
<td>( \Box(a \rightarrow \Box b) \land (\neg b \lor a) )</td>
<td>( b ) must be executed after ( a ), and ( a ) must precede ( b )</td>
<td>Y</td>
</tr>
<tr>
<td>Alt. Response</td>
<td><img src="image10" alt="Alt. Response" /></td>
<td>( \Box(a \rightarrow \Box (\neg a \lor b)) )</td>
<td>Every ( a ) must be followed by ( b ), without any other ( a ) inbetween</td>
<td>Y</td>
</tr>
<tr>
<td>Alt. Precedence</td>
<td><img src="image11" alt="Alt. Precedence" /></td>
<td>( (\neg b \lor a) \land \Box(b \rightarrow \Box (\neg b \lor a)) )</td>
<td>Every ( b ) must be preceded by ( a ), without any other ( b ) inbetween</td>
<td>Y</td>
</tr>
<tr>
<td>Alt. Succession</td>
<td><img src="image12" alt="Alt. Succession" /></td>
<td>( \Box(a \rightarrow \Box (\neg a \lor b)) \land (\neg b \lor a) \land \Box(b \rightarrow \Box (\neg b \lor a)) )</td>
<td>Combination of alternate response and alternate precedence</td>
<td>Y</td>
</tr>
<tr>
<td>Chain Response</td>
<td><img src="image13" alt="Chain Response" /></td>
<td>( \Box(a \rightarrow \Box b) )</td>
<td>If ( a ) is executed then ( b ) must be executed next</td>
<td>Y</td>
</tr>
<tr>
<td>Chain Precedence</td>
<td><img src="image14" alt="Chain Precedence" /></td>
<td>( \Box(b \rightarrow \Box a) )</td>
<td>Task ( b ) can be executed only immediately after ( a )</td>
<td>Y</td>
</tr>
<tr>
<td>Chain Succession</td>
<td><img src="image15" alt="Chain Succession" /></td>
<td>( \Box(a \equiv \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><img src="image16" alt="Not Coexistence" /></td>
<td>( \neg (\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><img src="image17" alt="Neg. Succession" /></td>
<td>( \Box(a \rightarrow \neg \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><img src="image18" alt="Neg. Chain Succession" /></td>
<td>( \Box(a \equiv \neg \Box b) )</td>
<td>Tasks ( a ) and ( b ) cannot be executed next to each other</td>
<td>Y</td>
</tr>
</tbody>
</table>

---

\(^2\) The full list of specifications is available here: [http://www.inf.unibz.it/~montali/AAAI14](http://www.inf.unibz.it/~montali/AAAI14)
description present in all papers about DECLARE, and shows the subtlety of directly adopting formulas originally devised in the infinite-trace setting to the one of finite traces. In fact, the same meaning is retained only for those formulas that are insensitive to infiniteness. Notice that the correct way of formalizing the intended meaning of negation chain succession on finite traces is \( \Box(a \equiv \neg b) \) (that is, \( \Box(a \equiv \neg \Box b) \)). This is equivalent to the other formulation in the infinite-trace setting, and actually it is insensitive to infiniteness.

Notice that there are several other DECLARE constraints, beyond standard patterns, that are not insensitive to infiniteness, such as \( \Box a \). Over infinite traces, \( \Box a \) states that \( a \) must be executed forever, whereas, on finite traces, it obviously stops requiring \( a \) when the trace ends.

5 Action Domains and Trajectories

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 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 \( \mathcal{P} \) into fluents \( \mathcal{F} \) and actions \( \mathcal{A} \), adding structural constraint (analogous to the DECLARE assumption) such as \( \Box(\bigwedge_{a \in \mathcal{A}} a) \land \Box(\bigwedge_{a \in \mathcal{A}} (a \rightarrow \bigwedge_{b \in \mathcal{A}, b \neq a} \neg b)) \), 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 \( \varphi_{\text{init}} \) involving only fluents, while effects can be modelled as:

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

where \( a \in \mathcal{A} \), 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 \( F(do(a, s)) \equiv \varphi^+ (s) \lor (F(s) \land \neg \varphi^- (s)) \) into the LTL\(_f\) formula:

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

In general, to specify effects we need special 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:

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

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:

\[\text{Theorem 7. Let } \varphi \text{ be an LTL}_f \text{ transition formula and } P \text{ any non-empty subset of } \mathcal{P}. \text{ Then all LTL}_f \text{ formulas of the form } \Box(\bigvee_{a \in \mathcal{P}} \varphi \rightarrow \varphi) \text{ are insensitive to infiniteness.}\]

\[\text{Proof. Suppose not. Then there exists a finite trace } \pi_f \text{ and a formula } \Box(\bigvee_{a \in \mathcal{P}} \varphi \rightarrow \varphi) \text{ such that } \pi_f \models \Box(\bigvee_{a \in \mathcal{P}} \varphi \rightarrow \varphi), \text{ but } \pi_f \models \neg(\bigvee_{a \in \mathcal{P}} \varphi). \text{ Hence, } \pi_f \models \Box(\bigvee_{a \in \mathcal{P}} \varphi \land \neg \varphi). \text{ That is there exist a point } i \text{ in the trace } \pi_f \text{ such that } \pi_f \models \Box(\bigvee_{a \in \mathcal{P}} \varphi \land \neg \varphi). \text{ Now observe that } i \text{ can only be in } \pi_f \text{ since in the } \{\text{end}\} \text{ part } \Box(\bigvee_{a \in \mathcal{P}} \varphi) \text{ is false. But then } \pi_f \models \Box(\bigvee_{a \in \mathcal{P}} \varphi \land \neg \varphi) \text{ contradicting the assumption.}\]

By applying the above theorem we can immediately show that (3) and (2) (for the latter, noting that it is equivalent to \( \Box(\varphi \rightarrow \Box (\varphi \lor \psi)) \)) are insensitive to infiniteness.

Also PDDL action effects (McDermott et al. 1998) can be encoded in LTL\(_f\), and show to be insensitive to infiniteness using the above theorem. Here, however, we focus on PDDL 3.0 trajectory constraints (Gerevini et al. 2009):

\[
(\text{at end } \phi) ::= \text{last } \land \phi \\
(\text{always } \phi) ::= \Box \phi \\
(\text{sometime } \phi) ::= \Box \phi \\
(\text{within } n \text{ } \phi) ::= \bigwedge_{0 \leq i \leq n} \phi \\
(\text{hold-after } n \text{ } \phi) ::= \bigwedge_{1 \leq i \leq n} \phi \\
(\text{hold-during } n_1 \text{ } n_2 \text{ } \phi) ::= \bigwedge_{i=n_1}^{n_2} \phi \\
(\text{at-most-once } \phi) ::= \bigwedge_{\phi \rightarrow \phi ^+} (\bigwedge_{1 \leq i \leq n_2} \phi \land \neg \phi_{i+1}) \\
(\text{sometime-after } \phi_1 \text{ } \phi_2) ::= \bigwedge_{\phi_1 \rightarrow \phi_2} \phi_2 \\
(\text{sometime-before } \phi_1 \text{ } \phi_2) ::= (\neg \phi_1 \land \neg \phi_2) \lor (\phi_1 \land \phi_2) \\
(\text{always-within } n \text{ } \phi) ::= \bigwedge_{\phi \rightarrow \phi ^+} (\bigwedge_{0 \leq i \leq n} \phi \land \neg \phi_{i+1})
\]

where \( \phi \) is a propositional formula on fluents, called goal formula. Most trajectory constraints are (variants) of DECLARE patterns, and we can ask if they are insensitive to infiniteness using Theorem 4. Moreover, the following general result holds. Let a goal formula be guarded when it is equivalent to \( (\forall F \in \mathcal{F}) \phi \land \phi \) any propositional formula. Then:

\[\text{Theorem 8. All trajectory constraints involving only guarded goal formulas, except from (always } \phi), \text{ are insensitive to infiniteness.}\]

6 Reasoning in LTL\(_f\) through NFAs

We can associate with each LTL\(_f\) formula \( \varphi \) an (exponential) NFA \( A_\varphi \) that accepts exactly the traces that make \( \varphi \) true. Various techniques for building such NFAs have been proposed in the literature, but they all require a detour to automata on infinite traces first. In (Bauer, Leucker, and Schallhart 2007) NFAs are used to check the compliance of an evolving trace to a formula expressed in LTL. The automaton construction is grounded on the one in (Lichtenstein, Pnueli, and Zuck 1985), which, by introducing past operators, focuses on finite traces. The procedure builds an NFA that recognizes both finite and infinite traces satisfying the formula. Such an automaton is indeed very similar to a generalized Büchi automaton (cf. the Büchi automaton construction for LTL formulas in (Baier, Katoen, and Guldemond Larsen 2008)).
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 LTLₙ semantics by dealing also with the next operator.

Here, we provide a simple direct algorithm for computing the NFA corresponding to an LTLₙ formula. The correctness of the algorithm is based on the fact that (i) we can associate with each LTLₙ formula φ a polynomial alternating automaton on words (AFW) Aφ that accept exactly the traces that make φ true (Di Giacomo and Vardi 2013), and (ii) every AFW can be transformed into an NFA, see, e.g., (Di Giacomo and Vardi 2013). However, to formulate the algorithm we do not need these notions, but we can work directly on the LTLₙ 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 LTLₙ formula can be transformed into negation normal form in linear time). We also assume P to include a special proposition last which denotes the last element of the trace. Note that last can be defined as last ⇔ • false. Then we define an auxiliary function δ that takes an LTLₙ formula ϕ (in negation normal form) and a propositional interpretation Π for P (including last), returning a positive boolean formula whose atoms are (quoted) ϕ subformulas.

\[
\begin{align*}
\delta(\text{¬}a, \Pi) &= \text{true} \text{ if } a \in \Pi \\
\delta(a, \Pi) &= \text{false} \text{ if } a \notin \Pi \\
\delta(\text{¬}a, \Pi) &= \text{false} \text{ if } a \notin \Pi \\
\delta(\varphi \land \varphi_2, \Pi) &= \delta(\varphi, \Pi) \land \delta(\varphi_2, \Pi) \\
\delta(\varphi \lor \varphi_2, \Pi) &= \delta(\varphi, \Pi) \lor \delta(\varphi_2, \Pi) \\
\delta(\text{¬} \varphi, \Pi) &= \delta(\varphi, \Pi) \\
\delta(\varphi \land \Pi, \Pi) &= \delta(\varphi, \Pi) \land \delta(\Pi, \Pi) \\
\delta(\varphi \lor \Pi, \Pi) &= \delta(\varphi, \Pi) \lor \delta(\Pi, \Pi) \\
\end{align*}
\]

Using function δ we can build the NFA Aφ of an LTLₙ formula ϕ in a forward fashion. States of Aφ are sets of atoms (recall that each atom is quoted ϕ subformulas) to be interpreted as a conjunction; the empty conjunction ∅ stands for true:

1: algorithm LTLₙ2NFA()
2: input LTLₙ formula φ
3: output NFA Aφ = (2ⁿ, S, \{s₀\}, ϕ, \{s_f\})
4: s₀ ← \{ϕ\} \triangleright single initial state
5: s₁ ← ∅ \triangleright single final state
6: S ← \{s₀, s₁\}, q ← ∅
7: while (S or q change) do
8: if (q ∈ S and q’ = \bigwedge_{(ϕ, ψ) ∈ q} δ(ϕ, S, Π)) then
9: S ← S ∪ \{q’\} \triangleright update set of states
10: q ← q ∪ \{(q, q, q')\} \triangleright update transition relation

where q’ is a set of quoted subformulas of ϕ that denotes a minimal interpretation such that q’ |= \bigwedge_{(ϕ, ψ) ∈ q} δ(ϕ, S, Π), but only the minimal ones. Notice that trivially we have (∅, a, ∅) ∈ q for every a ∈ Σ.

The algorithm LTLₙ2NFA terminates in at most exponential number of steps, and generates a set of states S whose size is at most exponential in the size of the formula ϕ.

**Theorem 9.** Let ϕ be an LTLₙ formula and Aφ the NFA constructed as above. Then π |= ϕ iff π ∈ L(Aφ) for any finite trace π.

**Proof (sketch).** Given a specific formula ϕ, δ grounded on the subformulas of ϕ becomes the transition function of the AFW, with initial state "ϕ" and no final states, corresponding to ϕ (De Giacomo and Vardi 2013). Then LTLₙ2NFA essentially transforms the AFW into a NFA.

Notice that above we have assumed to have a special proposition last ∈ P. If we want to remove such an assumption, we can easily transform the obtained automaton Aφ = (2ⁿ, S, \{ϕ\}, q, \{∅\}) into the new automaton A′φ = (2ⁿ\{last\}, S ∪ \{ended\}, \{ϕ\}, q, \{∅, ended\})

where: (q, q', q'') ∈ q' iff (q, q', q'') ∈ q, or (q, q' ∪ \{last\}) true ∈ q and q' = ended.

It is easy to see that the NFA obtained can be built on-the-fly while checking for nonemptiness, hence we have:

**Theorem 10.** Satisfiability of an LTLₙ formula can be checked in PSPACE by nonemptiness of Aφ (or A′φ).

Considering that validity and logical implications can be linearly reduced to satisfiability in LTLₙ (see Theorem 1), we can conclude the proposed construction is optimal wrt computational complexity for reasoning on LTLₙ.

We conclude this section by observing that using the obtained NFA (or in fact any correct NFA for LTLₙ in the literature, e.g., (Baier and McIlraith 2006)), one can easily check when the NFA obtained via the approach in (Edelkamp 2006; Gerevini et al. 2009) mentioned in the introduction, i.e., using directly the Büchi automaton for the formula, but by substituting the Büchi acceptance condition with the NFA one, is indeed correct, by simply checking language equivalence.

**7 Conclusions**

While the blurring between infinite and finite traces has been of help as a jump start, we should now sharpen our focus on LTL on finite traces (LTLₙ). This paper does it in two ways: by showing notable cases where the blurring does not harm (witnessed by insensitivity to infiniteness); and by proposing a direct route to develop algorithms for finite traces (as witnessed by the algorithm LTLₙ2NFA). Along the latter line, we note that LTLₙ2NFA can easily be extended to deal with the more powerful LDLₙ (Di Giacomo and Vardi 2013). In future work, we plan to investigate runtime monitoring (Bauer, Leucker, and Schallhart 2007) by using LTLₙ and LDLₙ monitors.

**Acknowledgments.** This research has been partially supported by the EU project Optique (FP7-IP-318338), and by the Sapienza Award 2013 “SPIRITLETS: SPIRITLET–based Smart spaces.”
References


This document is a copy of the accepted manuscript, published by AAAI Press Digital Library.


The final publication is available at http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8575