Temporal ASP: from logical foundations to practical use with `telingo`*

Pedro Cabalar[1 [0000−0001−7440−0953]]

University of Corunna, SPAIN
cabalar@udc.es

Abstract. This document contains some lecture notes for a seminar on Temporal Equilibrium Logic (TEL) and its application to Answer Set Programming (ASP) inside the 17th Reasoning Web Summer School (RW 2021). TEL is a temporal extension of ASP that introduces temporal modal operators as those from Linear-Time Temporal Logic. We present the basic definitions and intuitions for Equilibrium Logic and then extend these notions to the temporal case. We also introduce several examples using the temporal ASP tool `telingo`.

Keywords: Answer Set Programming · Linear Temporal Logic · Equilibrium Logic · Temporal Equilibrium Logic.

1 Introduction

Answer Set Programming [4] (ASP) is nowadays one of the most successful paradigms for declarative problem solving and practical Knowledge Representation. Based on the answer set (or stable model) semantics [17] for logic programs, ASP constitutes a declarative formalism and a natural choice for solving static combinatorial problems up to NP complexity (or Σ_2^P in the disjunctive case), but has also been applied to problems that involve a dynamic component and a higher complexity, like planning, well-known to be PSPACE-complete [5]. The use of ASP for temporal scenarios has been frequent since its early application for reasoning about actions and change [18]. Commonly, dynamic scenarios in ASP deal with transition systems and discrete time: instants are represented as integer values for a time-point parameter, added to all dynamic predicates. This temporal parameter is bound to a finite interval, from 0 to a maximum time-step n (usually called the *horizon*). Problems involving temporal search, such as planning or temporal explanation, are solved by multiple calls to the ASP solver and gradually increasing the horizon length.

Although this methodology is simple and provides a high degree of flexibility, it lacks for a differentiated treatment of temporal expressions (the time parameter is just one more logical variable to be grounded) and prevents the reuse of

* This research was partially supported by the Spanish Ministry of Economy and Competitivity (MINECO), grant TIN2017-84453-P.

the large corpora of techniques and results well-known from the temporal logic literature. In an attempt to overcome these limitations, the approach called *Temporal Equilibrium Logic* [7, 2] introduced a logical formalisation that combines ASP with modal temporal operators. This formalism constitutes an extension of *Equilibrium Logic* [21] which, in its turn, is a complete logical characterisation of (standard) ASP based on the intermediate logic of *Here-and-There* (HT) [20]. As a result, TEL is an expressive non-monotonic modal logic that shares the syntax of Linear-Time Temporal Logic (LTL) [22] but interprets temporal formulas under a non-monotonic semantics that properly extends stable models. This semantics is based on the idea of selecting some LTL temporal models of a theory Γ that satisfy some minimality condition, when examined under the weaker logic of temporal HT (THT). Thus, a temporal stable model of Γ is a kind of selected LTL model of Γ, and so, it has the form of a sequence of states, usually called a *trace*.

In the rest of this document we will first introduce some intuitions about Equilibrium Logic from a rule-based reasoning perspective and then shift to the definition of its temporal extension. After that, we will provide several examples and talk about their practical implementation in the ASP tool called `telingo`. Finally, we will close with some conclusions and open topics for future work or currently under study.

2 Rule-based reasoning and Equilibrium Logic

In this section, we partly reproduce the motivations included in [12] (see that paper for further detail). ASP is a rule-based paradigm sharing the same syntax as the logic programming language Prolog but with a different reading. Take a rule of the form:

```
smoke :- fire.
```
(1)

ASP uses a *bottom-up* reading of (1): "smoke is produced by fire." That is, whenever `fire` belongs to our current set of beliefs or certain facts, `smoke` must also be included in that set too. On the contrary, Prolog's *top-down* reading could be informally stated as "to obtain smoke, we need fire." That is, the rule describes a procedure to get `smoke` as a goal which consists in pursuing `fire` as a new goal. Regardless of the application direction, it seems clear that rules have a conditional form with a right-hand condition (*body*) and a left-hand consequent (*head*) that in our example (1) respectively correspond to `fire` and `smoke`. Thus, a straightforward logical formalisation would be understanding (1) as the implication $\mathit{fire} \rightarrow \mathit{smoke}$ in classical propositional logic. This guarantees, for instance, that if we add *fire* as a program fact, we will get *smoke* as a conclusion (by application of *modus ponens*). So, the "operational" aspect of rule (1) can be captured by classical implication. However, the *semantics* of a classical implication is not enough to cover the intuitive meaning of a program rule. If our program only contains (1) and we read it as a rule, it is clear that *fire* is not satisfied, since *no rule can yield* that atom, and so, *smoke* is not obtained

either. However, implication *fire* $\rightarrow$ *smoke*, which amounts to the classically equivalent disjunction $\neg$*fire* $\vee$ *smoke*, has three classical models: $\emptyset$ (both atoms false), $\{$*smoke*$\}$ and $\{$*fire*, *smoke*$\}$. Note that the two last models seem to consider situations in which *smoke* or *fire* could be arbitrarily *assumed as true*, even though the program provides *no way to prove them*. An important observation is that $\emptyset$ happens to be the smallest model (with respect to set inclusion). This model is interesting because, somehow, it reflects the principle of not adding arbitrary true atoms that we are not forced to believe, and it coincides with the expected meaning for a program just containing (1). The existence of a least classical model is, in fact, guaranteed for logic programs without negation (or disjunction), so-called *positive logic programs*, and so, it was adopted as the main semantics [14] for logic programming until the introduction of negation. However, when negation came into play, classical logic was revealed to be insufficient again, even under the premise of minimal models selection. Suppose we have a program Π_1 consisting of the rules:

```
fill :- empty, not fire.        (2)
empty.                          (3)
```

where (2) means that we always *fill* our gas tank if it is *empty* and there is no evidence on *fire*, and (3) says that the tank is empty indeed. As before, *fire* cannot be proved (it is not head of any rule) and so, the condition of (2) is satisfied, producing *fill* as a result. The straightforward logical translation of (2) is *empty* $\wedge$ $\neg$*fire* $\rightarrow$ *fill* that, in combination with fact (3), produces three models: $T_1 = \{$*empty*, *fill*$\}$, $T_2 = \{$*empty*, *fire*$\}$ and $T_3 = \{$*empty*, *fire*, *fill*$\}$. Unfortunately, there is no least classical model any more: both T_1 (the expected model) and T_2 are minimal with respect to set inclusion. After all, the previous implication is classically *equivalent* to *empty* $\rightarrow$ *fire* $\vee$ *fill* which does not capture the directional behaviour of rule (2). The undesired minimal model T_2 is assuming *fire* to be true, although there is no way to prove that fact in the program. So, apparently, classical logic is too weak for capturing the meaning of logic programs in the sense that it provides the expected model(s), but also accepts other models (like T_2 and T_3) in which some atoms are abitrarily assumed to be true but not "justified by the program."

Suppose we had a way to classify true atoms distinguishing between those just being an *assumption* (classical model T) and those being also *justified* or *proved* by program rules. In our intended models, the set of justified atoms should precisely coincide with the set of assumed ones in T. As an example, suppose our assumed atoms are $T_3 = \{$*empty*, *fire*, *smoke*$\}$. Any justification should include *empty* because of fact (3). However, rule (2) seems to be unapplicable, because we are currently assuming that *fire* is possibly true, *fire* $\in T_3$, and so '`not fire`' is not acceptable – there is some (weak) evidence about fire. As a result, atom *fill* is not necessarily justified and we can only derive $\{$*empty*$\}$, which is strictly smaller than our initial assumption T_3. Something similar happens for assumption $T_2 = \{$*empty*, *fire*$\}$. If we take classical model $T_1 = \{$*empty*, *fill*$\}$ instead as an initial assumption, then the body of rule (2) becomes applicable,

since no evidence on *fire* can be found, that is, $\mathit{fire} \notin T_1$. As a result, the justified atoms are now $\{\mathit{empty}, \mathit{fill}\} = T_1$ and the classical model T_1 becomes the unique intended (stable) model of the program.

The method we have just used with the example can be seen as an informal description of the original definition of the stable models semantics [17]. This definition consisted of classical logic reinforced with an *extra-logical* program transformation (for interpreting negation) and then using application of rules to obtain the actually derived or justified information. We show next how it is possible to provide an equivalent definition that exclusively resorts to logical concepts but using a different underlying formalism, weaker than classical logic.

Although, as we have seen, our interest is focused on rules, the semantics of Equilibrium Logic [21] can be defined on any arbitrary propositional formula. Covering arbitrary formulas is, in fact, simpler and more homogeneous than the original definition of stable models based on the reduct syntactic transformation. Equilibrium models are defined by a models selection criterion on top of the intermediate logic of Here-and-There (HT) [20], stronger than intuitionistic logic but weaker than classical logic. The latter can be seen as a three-valued logic where an atom can be false, assumed or proved, as we discussed before. Formally, an HT *interpretation* is a pair of sets of atoms $\langle H, T\rangle$ satisfying $H \subseteq T$ so that: any atom $p \in H$ is considered as *proved* or *justified*; an atom $p \in T$ is considered as *assumed*; and any atom $p \notin T$ is understood as *false*. As we can see, proved implies assumed, that is, the set of justified atoms H is always a subset of the assumed ones T. Intuitively, T acts as our "initial assumption" while the subset H contains those atoms from T currently considered as justified. Let At be the collection of all atomic formulas in our given language. Then $H \subseteq T \subseteq At$ and all atoms in $At \setminus T$ are considered false in this model. An HT interpretation $\langle H, T\rangle$ is said to be *total* when $H = T$ (that is, when all assumptions are justified).

As we did for atoms, formulas can also be considered to be false, assumed or proved. We will use a satisfaction relation $\langle H, T\rangle \models \varphi$ to represent that $\langle H, T\rangle$ makes formula φ to be proved or justified. Sometimes, however, we may happen that this relation does not hold $\langle H, T\rangle \not\models \varphi$ while in classical logic satisfaction $T \models \varphi$ using the assumptions in T is true. Then, we may say that the formula is just assumed. Finally, when φ is not even classically satisfied by T, $T \not\models \varphi$, we can guarantee that the formula is false. Formally, the fact that an interpretation $\langle H, T\rangle$ *satisfies* a formula φ (or makes it justified), written $\langle H, T\rangle \models \varphi$, is recursively defined as follows:

- $\langle H, T\rangle \not\models \bot$
- $\langle H, T\rangle \models p$ iff $p \in H$
- $\langle H, T\rangle \models \varphi \wedge \psi$ iff $\langle H, T\rangle \models \varphi$ and $\langle H, T\rangle \models \psi$
- $\langle H, T\rangle \models \varphi \vee \psi$ iff $\langle H, T\rangle \models \varphi$ or $\langle H, T\rangle \models \psi$
- $\langle H, T\rangle \models \varphi \rightarrow \psi$ iff both (i) $T \models \varphi$ implies $T \models \psi$ and (ii) $\langle H, T\rangle \models \varphi$ implies $\langle H, T\rangle \models \psi$

By abuse of notation, we use '$\models$' both for classical and for HT-satisfaction: the ambiguity is resolved by the form of the left interpretation (a single set T for classical and a pair $\langle H, T\rangle$ for HT). We say that an interpretation $\langle H, T\rangle$ is a

model of a theory (set of formulas) Γ iff $\langle H, T\rangle \models \varphi$ for all $\varphi \in \Gamma$. We say that a propositional theory Γ *entails* some formula φ, written $\Gamma \models \varphi$, if any model of Γ is also a model of φ.

As we can see, everything is pretty standard excepting for the interpretation of implication, which imposes a stronger condition than in classical logic. In order to satisfy $\langle H, T\rangle \models \varphi \rightarrow \psi$, the standard condition would be (ii), that is, if the antecedent holds, then the consequent must hold too. In our case, the reading is closer to an application of *modus ponens* in an inference rule: if the antecedent is proved, then we can also prove the consequent. This condition, however, is further reinforced by (i) which informally means that our set of assumptions T classically satisfy the implication $\varphi \rightarrow \psi$ as well.

The following proposition tells us that satisfaction for total models amounts to classical satisfaction:

Proposition 1. *For any formula φ and set of atoms T, $\langle T, T\rangle \models \varphi$ iff $T \models \varphi$ in classical logic.*

We may read this result saying that classical models are a subset of HT models (they correspond to total HT models). This immediately means that any HT tautology is also a classical tautology. The opposite does not hold, namely, there are classical tautologies that are not HT tautologies. We will see later several examples.

Classical satisfaction for T allows us to keep the three-valued reading (*false*, *assumed* or *proved*) also for formulas in the following way. We say that $\langle H, T\rangle$ makes formula φ:

- *proved* when $\langle H, T\rangle \models \varphi$,
- *assumed* when $T \models \varphi$,
- *false* when $T \not\models \varphi$.

Interestingly, as happened with atoms, formulas also satisfy that anything proved must also be assumed. This is stated as the following property called *persistence*:

Proposition 2 (Persistence). *For any formula φ and any HT interpretation $\langle H, T\rangle$ we can show that, if φ is proved then it is also assumed, that is: $\langle H, T\rangle \models \varphi$ implies $T \models \varphi$.*

Notice that we did not provide satisfaction of negation $\neg\varphi$. This is because negation is not included above because it can be defined in terms of implication as the formula $\varphi \rightarrow \bot$, as happens in intuitionistic logic. Using that abbreviation and after some analysis, it can be proved that $\langle H, T\rangle \models \neg\varphi$ amounts to $T \not\models \varphi$, that is, $\neg\varphi$ is justified simply when φ is not assumed, that is, when it is false. Apart from negation, we also define the common Boolean operators $\top \stackrel{\text{def}}{=} \neg\bot$ and $\varphi \leftrightarrow \psi \stackrel{\text{def}}{=} (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$.

If we apply the persistence property to our example program Π_1, this means that any of its models $\langle H, T\rangle \models \Pi$ must satisfy $T \models \Pi$ as well. As we saw, we have only three possibilities for the latter, T_1, T_2 and T_3. On the other hand, the program fact (3) fixes $\mathit{empty} \in H$. Now, take assumption $T_1 = \{\mathit{empty}, \mathit{fill}\}$. The

only model we get is $\langle T_1, T_1\rangle$ because the other possible subset $H = \{\mathit{empty}\}$ of T_1 does not satisfy (2): *empty* is justified, *fire* is false, so we should get *fill*. Take T_2 instead. Apart from $\langle T_2, T_2\rangle$, in this case we also get a model $\langle H, T_2\rangle$ with $H = \{\mathit{empty}\}$. In such a case, *fire* is only assumed true, but not proved. As a result, the rule is satisfied because its condition $\neg \mathit{fire}$ is false (we have some evidence on *fire*) and so $\langle \{\mathit{empty}\}, T_2\rangle$ becomes a model. This is a clear evidence that our initial assumption adding *fire* is not necessarily proved when we check the program rules. In the case of $T_3 = \{\mathit{empty}, \mathit{fire}, \mathit{fill}\}$ we have a similar situation. Interpretations with $H = \{\mathit{empty}\}$, $H = \{\mathit{empty}, \mathit{fire}\}$ or $H = \{\mathit{empty}, \mathit{fill}\}$ are also models. Note that in all of them, the only atom that is always proved is *empty*, pointing out again that *fire* or *fill* are not necessarily justified (cannot be proved using the program rules).

It must be understood that, at this point, the tag "justified" or "proved" just refers to a second kind of truth, stronger than "assumed." This tag will only acquire a real "provability" meaning once we introduce a models minimisation. For instance, for the formula $\mathit{fire} \rightarrow \mathit{smoke}$, we will have a model $H = T = \{\mathit{smoke}\}$ where *smoke* is being considered justified. We still miss some minimisation criterion to consider justified or proved only those atoms and formulas that we are certain to be so. This minimisation selects some particular HT models and will follow the intuitive idea: given a fixed set of assumptions T, minimise proved atoms H. If we additionally require that anything assumed must be eventually proved, we get the following definition of *equilibrium models* first introduced by Pearce [21]:

Definition 1 (Equilibrium Model). *A total HT interpretation $\langle T, T\rangle$ is an* equilibrium model *of a theory Γ if $\langle T, T\rangle \models \Gamma$ and there is no $H \subset T$ such that $\langle H, T\rangle \models \Gamma$. When this happens, we also say that T is a* stable model *(or answer set) of Γ.*

From the logical point of view, it is now an easy task to define a stable model. The intuition is that we will be interested in cases where anything assumed true in set T eventually becomes necessarily proved, ie , $H = T$ is the only possibility for assumption T.

Back to our example, the only stable model of Π_1 is the expected $T_1 = \{\mathit{empty}, \mathit{fill}\}$. This is because for the other two classical models $T_2 = \{\mathit{empty}, \mathit{fire}\}$ and $T_3 = \{\mathit{empty}, \mathit{fill}, \mathit{fire}\}$ we could see in the previous section that there were smaller sets H' that formed possible models of the program such as $\langle \{\mathit{empty}\}, T_2\rangle$ or $\langle \{\mathit{empty}\}, T_3\rangle$. In the case of T_1, however, the only obtained model is $\langle T_1, T_1\rangle$ and no smaller $H \subset T_1$ can be used to form a model.

By selecting the equilibrium models, we obtain a non-monotonic entailment relation, that is, we may obtain conclusions that, after we add new information, can be retracted. For instance, in our example, if we are now said that *fire* has been observed and we take program $\Pi_2 = \Pi_1 \cup \{\mathit{fire}\}$, the classical models become $T_4 = \{\mathit{empty}, \mathit{fire}\}$ and $T_5 = \{\mathit{empty}, \mathit{fire}, \mathit{fill}\}$. Clearly, T_4 will be a stable model, since there is no way to remove any of the two atoms *empty*, *fire* that are now facts in program Π_2. However, T_5 is not in equilibrium, since we can

form $H = \{empty, fire\}$ and $\langle H, T_5\rangle \models \Pi_2$ because rule (2) is always satisfied, as its body is falsified since $T_5 \not\models \neg fire$. From the only stable model we obtain, we conclude that we cannot *fill* the tank any more, while this atom was among the previous conclusions when we had no information about *fire*.

As said before, some classical tautologies are not HT tautologies. An interesting example is the law of excluded middle $a \vee \neg a$. Intuitively, this formula means that a has to be proved or assumed and, accordingly, it has the HT countermodel $\langle \emptyset, \{a\}\rangle$. Therefore, if we include $a \vee \neg a$ for some atom in our theory, we force that assuming a is enough to consider it proved, so p will somehow behave "classically." Moreover, if we add the formula $a \vee \neg a$ for all atoms in the signature At , then all models are forced to be total $\langle T, T\rangle$ and Equilibrium Logic collapses into classical logic.

3 Temporal Equilibrium Logic

An important advantage of the definition of stable models based on Equilibrium Logic is that it provides a purely logical characterisation that has no syntactic limitations (it applies to arbitrary propositional formulas) and is easy to extend with the incorporation of new constructs or the definition of new combined logics. As happened with Equilibrium Logic, the definition of *(Linear-time) Temporal Equilibrium Logic* (TEL) is done in two steps. First, we define a monotonic temporal extension of HT, called *(Linear-time) Temporal Here-and-There* (THT) and, second, we select some models from THT that are said to be *in equilibrium*, obtaining in this way a non-monotonic entailment relation.

We reproduce next part of the contents from [1] and [10]. The original definition of TEL was thought as a direct non-monotonic extension of standard LTL, so that models had the form of infinite traces. However, this rules out computation by ASP technology and is unnatural for applications like planning, where plans amount to finite prefixes of one or more traces [13]. In a recent line of research [11], TEL was extended to cope with finite traces (which are closer to ASP computation). On the one hand, this amounts to a restriction of THT and TEL to finite traces. On the other hand, this is similar to the restriction of LTL to LTL_f advocated by [13]. Our new approach, dubbed TEL_f, has the following advantages. First, it is readily implementable via ASP technology. Second, it can be reduced to a normal form which is close to logic programs and much simpler than the one obtained for TEL. Finally, its temporal models are finite and offer a one-to-one correspondence to plans. Interestingly, TEL_f also sheds light on concepts and methodology used in incremental ASP solving when understanding incremental parameters as time points. Another distinctive feature of TEL_f is the inclusion of future as well as past temporal operators. When using the causal reading of program rules, it is generally more natural to draw upon the past in rule bodies and to refer to the future in rule heads. As well, past operators are much easier handled computationally than their future counterparts when it comes to incremental reasoning, since they refer to already computed knowledge.

In what follows, we present the general logics THT (monotonic) and TEL (non-monotonic) allowing traces of any length (possibly infinite), and will later on use the subindices ω or f to denote the particular cases where traces are always infinite or always finite, respectively. The syntax of THT (and TEL) is the same as for LTL with past operators. Given a (countable, possibly infinite) set At of propositional variables (called *alphabet*), *temporal formulas* φ are defined by the grammar:

$$\varphi ::= a \mid \bot \mid \varphi_1 \otimes \varphi_2 \mid \bullet\varphi \mid \varphi_1 \,\mathbf{S}\, \varphi_2 \mid \varphi_1 \,\mathbf{T}\, \varphi_2 \mid \circ\varphi \mid \varphi_1 \,\mathbb{U}\, \varphi_2 \mid \varphi_1 \,\mathbb{R}\, \varphi_2 \mid \varphi_1 \,\mathbb{W}\, \varphi_2$$

where $a \in At$ is an atom and $\otimes$ is any binary Boolean connective $\otimes \in \{\rightarrow, \wedge, \vee\}$. The last six cases correspond to the temporal connectives whose names are listed below:

Past		*Future*	
	$\bullet$ for *previous*		$\circ$ for *next*
	$\mathbf{S}$ for *since*		$\mathbb{U}$ for *until*
	$\mathbf{T}$ for *trigger*		$\mathbb{R}$ for *release*
			$\mathbb{W}$ for *while*

We also define several common derived temporal operators:

$\blacksquare\varphi \stackrel{\text{def}}{=} \bot \,\mathbf{T}\, \varphi$	*always before*	$\Box\varphi \stackrel{\text{def}}{=} \bot \,\mathbb{R}\, \varphi$	*always afterward*
$\blacklozenge\varphi \stackrel{\text{def}}{=} \top \,\mathbf{S}\, \varphi$	*eventually before*	$\Diamond\varphi \stackrel{\text{def}}{=} \top \,\mathbb{U}\, \varphi$	*eventually afterward*
$\mathbf{I} \stackrel{\text{def}}{=} \neg\bullet\top$	*initial*	$\mathbb{F} \stackrel{\text{def}}{=} \neg\circ\top$	*final*
$\widehat{\bullet}\varphi \stackrel{\text{def}}{=} \bullet\varphi \vee \mathbf{I}$	*weak previous*	$\widehat{\circ}\varphi \stackrel{\text{def}}{=} \circ\varphi \vee \mathbb{F}$	*weak next*

A *(temporal) theory* is a (possibly infinite) set of temporal formulas. Note that we use solid operators to refer to the past, while future-time operators are denoted by outlined symbols.

As happens with HT with respect to classical logic, logics THT and LTL share the same syntax but, they have a different semantics, the former being a weaker logic than the latter. The semantics of LTL relies on the concept of a *trace*, a (possibly infinite) sequence of *states*, each of which is a set of atoms. For defining traces, we start by introducing some notation to deal with intervals of integer time points. Given $a \in \mathbb{N}$ and $b \in \mathbb{N} \cup \{\omega\}$, we let $[a..b]$ stand for the set $\{i \in \mathbb{N} \mid a \leq i \leq b\}$, $[a..b)$ for $\{i \in \mathbb{N} \mid a \leq i < b\}$ and $(a..b]$ for $\{i \in \mathbb{N} \mid a < i \leq b\}$. In LTL, a *trace* $\mathbf{T}$ of length λ over alphabet At is a sequence $\mathbf{T} = (T_i)_{i \in [0..\lambda)}$ of sets $T_i \subseteq At$. We sometimes use the notation $|\mathbf{T}| \stackrel{\text{def}}{=} \lambda$ to stand for the length of the trace. We say that $\mathbf{T}$ is *infinite* if $|\mathbf{T}| = \omega$ and *finite* if $|\mathbf{T}| \in \mathbb{N}$. To represent a given trace, we write a sequence of sets of atoms concatenated with '$\cdot$'. For instance, the finite trace $\{a\} \cdot \emptyset \cdot \{a\} \cdot \emptyset$ has length 4 and makes a true at even time points and false at odd ones. For infinite traces, we sometimes use ω-regular expressions like, for instance, in the infinite trace $(\{a\} \cdot \emptyset)^\omega$ where all even positions make a true and all odd positions make it false.

A state i is represented as a pair of sets of atoms $\langle H_i, T_i \rangle$ with $H_i \subseteq T_i \subseteq At$ where H_i (standing for "here") contains the proved atoms, whereas T_i (standing

for "there") contains the assumed atoms. On the other hand, false atoms are just the ones not assumed, captured by $At \setminus T_i$. An HT-trace of length λ over alphabet At is a sequence of pairs $(\langle H_i, T_i\rangle)_{i\in[0..\lambda)}$ with $H_i \subseteq T_i$ for any $i \in [0..\lambda)$. For convenience, we usually represent the HT-trace as the pair $\langle \mathbf{H}, \mathbf{T}\rangle$ of traces $\mathbf{H} = (H_i)_{i\in[0..\lambda)}$ and $\mathbf{T} = (T_i)_{i\in[0..\lambda)}$. Given $\mathbf{M} = \langle \mathbf{H}, \mathbf{T}\rangle$, we also denote its length as $|\mathbf{M}| \stackrel{\text{def}}{=} |\mathbf{H}| = |\mathbf{T}| = \lambda$. Note that the two traces $\mathbf{H}$, $\mathbf{T}$ must satisfy a kind of order relation, since $H_i \subseteq T_i$ for each time point i. Formally, we define the ordering $\mathbf{H} \leq \mathbf{T}$ between two traces of the same length λ as $H_i \subseteq T_i$ for each $i \in [0..\lambda)$. Furthermore, we define $\mathbf{H} < \mathbf{T}$ as both $\mathbf{H} \leq \mathbf{T}$ and $\mathbf{H} \neq \mathbf{T}$. Thus, an HT-trace can also be defined as any pair $\langle \mathbf{H}, \mathbf{T}\rangle$ of traces such that $\mathbf{H} \leq \mathbf{T}$. The particular type of HT-traces satisfying $\mathbf{H} = \mathbf{T}$ are called *total*.

Given any HT-trace $\mathbf{M} = \langle \mathbf{H}, \mathbf{T}\rangle$, we define the THT satisfaction of formulas as follows.

Definition 2 (THT-satisfaction). *An HT-trace $\mathbf{M} = \langle \mathbf{H}, \mathbf{T}\rangle$ of length λ over alphabet At* satisfies *a temporal formula φ at time point $k \in [0..\lambda)$, written $\mathbf{M}, k \models \varphi$, if the following conditions hold:*

1. $\mathbf{M}, k \models \top$ *and* $\mathbf{M}, k \not\models \bot$
2. $\mathbf{M}, k \models a$ *if* $a \in H_k$ *for any atom* $a \in At$
3. $\mathbf{M}, k \models \varphi \wedge \psi$ *iff* $\mathbf{M}, k \models \varphi$ *and* $\mathbf{M}, k \models \psi$
4. $\mathbf{M}, k \models \varphi \vee \psi$ *iff* $\mathbf{M}, k \models \varphi$ *or* $\mathbf{M}, k \models \psi$
5. $\mathbf{M}, k \models \varphi \rightarrow \psi$ *iff* $\langle \mathbf{H}', \mathbf{T}\rangle, k \not\models \varphi$ *or* $\langle \mathbf{H}', \mathbf{T}\rangle, k \models \psi$, *for all* $\mathbf{H}' \in \{\mathbf{H}, \mathbf{T}\}$
6. $\mathbf{M}, k \models \bullet\varphi$ *iff* $k > 0$ *and* $\mathbf{M}, k{-}1 \models \varphi$
7. $\mathbf{M}, k \models \varphi \,\mathsf{S}\, \psi$ *iff for some* $j \in [0..k]$, *we have* $\mathbf{M}, j \models \psi$ *and* $\mathbf{M}, i \models \varphi$ *for all* $i \in (j..k]$
8. $\mathbf{M}, k \models \varphi \,\mathsf{T}\, \psi$ *iff for all* $j \in [0..k]$, *we have* $\mathbf{M}, j \models \psi$ *or* $\mathbf{M}, i \models \varphi$ *for some* $i \in (j..k]$
9. $\mathbf{M}, k \models \circ\varphi$ *iff* $k + 1 < \lambda$ *and* $\mathbf{M}, k{+}1 \models \varphi$
10. $\mathbf{M}, k \models \varphi \,\mathbb{U}\, \psi$ *iff for some* $j \in [k..\lambda)$, *we have* $\mathbf{M}, j \models \psi$ *and* $\mathbf{M}, i \models \varphi$ *for all* $i \in [k..j)$
11. $\mathbf{M}, k \models \varphi \,\mathbb{R}\, \psi$ *iff for all* $j \in [k..\lambda)$, *we have* $\mathbf{M}, j \models \psi$ *or* $\mathbf{M}, i \models \varphi$ *for some* $i \in [k..j)$
12. $\mathbf{M}, k \models \varphi \,\mathbb{W}\, \psi$ *iff for all* $j \in [k..\lambda)$, *we have* $\langle \mathbf{H}', \mathbf{T}\rangle, j \models \varphi$ *or* $\langle \mathbf{H}', \mathbf{T}\rangle, i \not\models \psi$ *for some* $i \in [k..j)$ *and for all* $\mathbf{H}' \in \{\mathbf{H}, \mathbf{T}\}$

□

In general, these conditions inherit the interpretation of connectives from LTL (with past operators) with just a few differences. A first minor variation is that we allow traces of arbitrary length λ, including both infinite ($\lambda = \omega$) and finite ($\lambda \in \mathbb{N}$) traces. A second difference with respect to LTL is the new connective $\varphi \,\mathbb{W}\, \psi$ which is also a kind of temporally-iterated HT implication. Its intuitive reading is "keep doing φ while condition ψ holds." In LTL, $\varphi \mathbb{W} \psi$ would just amount to $\neg\psi \,\mathbb{R}\, \varphi$, but under HT semantics both formulas have a different meaning, as the latter may provide evidence for φ even though the condition ψ does not hold.

An HT-trace $\mathbf{M}$ is a *model* of a temporal theory Γ if $\mathbf{M}, 0 \models \varphi$ for all $\varphi \in \Gamma$. We write $THT(\Gamma, \lambda)$ to stand for the set of THT-models of length λ of a theory Γ, and define $THT(\Gamma) \stackrel{\text{def}}{=} THT(\Gamma, \omega) \cup \bigcup_{\lambda \in \mathbb{N}} THT(\Gamma, \lambda)$. That is, $THT(\Gamma)$ is the whole set of models of Γ of any length. For $\Gamma = \{\varphi\}$, we just write $THT(\varphi, \lambda)$ and $THT(\varphi)$. We can analogously define $LTL(\Gamma, \lambda)$, that is, the set of traces of length λ that satisfy theory Γ, and $LTL(\Gamma)$, that is, the LTL-models of Γ any length. We omit specifying LTL satisfaction since it coincides with THT when HT-traces are total.

Proposition 3 ([2, 11]). *Let* $\mathbf{T}$ *be a trace of length* λ*,* φ *a temporal formula, and* $k \in [0..\lambda)$ *a time point.*

Then, $\mathbf{T}, k \models \varphi$ *in LTL iff* $\langle \mathbf{T}, \mathbf{T} \rangle, k \models \varphi$. □

In fact, total models can be forced by adding the following set of *excluded middle* axioms:

$$\Box(a \vee \neg a) \qquad \text{for each atom } a \in At \quad \text{in the signature.} \tag{EM}$$

Proposition 4 ([2, 11]). *Let* $\langle \mathbf{H}, \mathbf{T} \rangle$ *be an HT-trace and (EM) the theory containing all excluded middle axioms for every atom* $a \in At$ *. Then,* $\langle \mathbf{H}, \mathbf{T} \rangle$ *is a model of (EM) iff* $\mathbf{H} = \mathbf{T}$. □

Satisfaction of derived operators can be easily deduced, as shown next.

Proposition 5 ([2, 11]). *Let* $\mathbf{M} = \langle \mathbf{H}, \mathbf{T} \rangle$ *be an HT-trace of length* λ *over At . Given the respective definitions of derived operators, we get the following satisfaction conditions:*

12. $\mathbf{M}, k \models \mathsf{I}$ *iff* $k = 0$
13. $\mathbf{M}, k \models \widehat{\bullet}\varphi$ *iff* $k = 0$ *or* $\mathbf{M}, k{-}1 \models \varphi$
14. $\mathbf{M}, k \models \blacklozenge\varphi$ *iff* $\mathbf{M}, i \models \varphi$ *for some* $i \in [0..k]$
15. $\mathbf{M}, k \models \blacksquare\varphi$ *iff* $\mathbf{M}, i \models \varphi$ *for all* $i \in [0..k]$
16. $\mathbf{M}, k \models \mathbb{F}$ *iff* $k + 1 = \lambda$
17. $\mathbf{M}, k \models \widehat{\bigcirc}\varphi$ *iff* $k + 1 = \lambda$ *or* $\mathbf{M}, k{+}1 \models \varphi$
18. $\mathbf{M}, k \models \Diamond\varphi$ *iff* $\mathbf{M}, i \models \varphi$ *for some* $i \in [k..\lambda)$
19. $\mathbf{M}, k \models \Box\varphi$ *iff* $\mathbf{M}, i \models \varphi$ *for all* $i \in [k..\lambda)$

□

Given a set of THT-models, we define the ones in equilibrium as follows.

Definition 3 (Temporal Equilibrium/Stable Model). *Let* $\mathfrak{S}$ *be some set of HT-traces. A total HT-trace* $\langle \mathbf{T}, \mathbf{T} \rangle \in \mathfrak{S}$ *is a* temporal equilibrium model *of* $\mathfrak{S}$ *iff there is no other* $\mathbf{H} < \mathbf{T}$ *such that* $\langle \mathbf{H}, \mathbf{T} \rangle \in \mathfrak{S}$*. The trace* $\mathbf{T}$ *is called a* temporal stable model *(TS-model) of* $\mathfrak{S}$*.* □

We further talk about temporal equilibrium or temporal stable models of a theory Γ when $\mathfrak{S} = THT(\Gamma)$, respectively. Moreover, we write $TEL(\Gamma, \lambda)$ and $TEL(\Gamma)$ to stand for the temporal equilibrium models of $THT(\Gamma, \lambda)$ and $THT(\Gamma)$ respectively. We write $TSM(\Gamma, \lambda)$ and $TSM(\Gamma)$ to stand for the corresponding sets of TS-models. One interesting observation is that, since temporal

equilibrium models are total models $\langle \mathbf{T}, \mathbf{T} \rangle$, due to Proposition 3, we obtain $TSM(\Gamma, \lambda) \subseteq LTL(\Gamma, \lambda)$ that is, temporal stable models are a subset of LTL-models.

Temporal Equilibrium Logic (TEL) is the (non-monotonic) logic induced by temporal equilibrium models. We can also define the variants TEL_ω and TEL_f by applying the corresponding restriction to infinite and finite traces, respectively.

As an example of non-monotonicity, consider the formula

$$\Box(\bullet \mathit{loaded} \wedge \neg \mathit{unloaded} \rightarrow \mathit{loaded}) \tag{4}$$

that corresponds to the inertia for *loaded*, together with the fact *loaded*, describing the initial state for that fluent. Without entering into too much detail, this formula behaves as the logic program with the rules:

```
loaded(0).
loaded(T) :- loaded(T-1), not unloaded(T).
```

for any time point T>0. As expected, for some fixed λ, we get a unique temporal stable model of the form $\{\mathit{loaded}\}^\lambda$. This entails that *loaded* is always true, viz. $\Box \mathit{loaded}$, as there is no reason for *unloaded* to become true. Note that in the most general case of TEL, we actually get one stable model per each possible λ, including $\lambda = \omega$. Now, consider formula (4) along with $\mathit{loaded} \wedge \bigcirc\bigcirc \mathit{unloaded}$ which amounts to adding the fact unloaded(2). As expected, for each λ, the only temporal stable model now is $\mathbf{T} = \{\mathit{loaded}\} \cdot \{\mathit{loaded}\} \cdot \{\mathit{unloaded}\} \cdot \emptyset^\alpha$ where α can be $*$ or ω. Note that by making $\bigcirc\bigcirc \mathit{unloaded}$ true, we are also forcing $|\mathbf{T}| \geq 3$, that is, there are no temporal stable models (nor even THT-models) of length smaller than three. Thus, by adding the new information $\bigcirc\bigcirc \mathit{unloaded}$ some conclusions that could be derived before, such as $\Box \mathit{loaded}$, are not derivable any more.

As an example emphasizing the behavior of finite traces, take the formula

$$\Box(\neg a \rightarrow \bigcirc a) \tag{5}$$

which can be seen as a program rule "a(T+1) :− not a(T)" for any natural number T. As expected, temporal stable models make a false in even states and true in odd ones. However, we cannot take finite traces making a false at the final state $\lambda - 1$, since the rule would force $\bigcirc a$ and this implies the existence of a successor state. As a result, the temporal stable models of this formula have the form $(\emptyset \cdot \{a\})^+$ for finite traces in TEL_f, or the infinite trace $(\emptyset \cdot \{a\})^\omega$ in TEL_ω.

Another interesting example is the temporal formula

$$\Box(\neg\bigcirc a \rightarrow a) \wedge \Box(\bigcirc a \rightarrow a).$$

The corresponding rules "a(T) :− not a(T+1)" and "a(T) :− a(T+1)" have no stable model [15] when grounded for all natural numbers T. This is because there is no way to build a finite proof for any a(T), as it depends on infinitely many next states to be evaluated. The same happens in TEL_ω, that is, we get no infinite temporal stable model. However in TEL_f, we can use the fact that

$\circ a$ is always false in the last state. Then, $\Box(\neg\circ a \rightarrow a)$ supports a in that state and therewith $\Box(\circ a \rightarrow a)$ inductively supports a everywhere.

As an example of a temporal expression not so close to logic programming, consider the formula $\Box\Diamond a$, which is normally used in LTL_ω to assert that a occurs infinitely often. As discussed by [13], if we assume finite traces, then the formula collapses to $\Box(\mathbb{F} \rightarrow a)$ in LTL_f, that is, a is true at the final state (and either true or false everywhere else). The same behavior is obtained in THT_ω and THT_f, respectively. However, if we move to TEL, a truth minimization is additionally required. As a result, in TEL_f, we obtain a unique temporal stable model for each fixed $\lambda \in \mathbb{N}$, in which a is true at the last state, and false everywhere else. Unlike this, TEL_ω yields no temporal stable model at all. This is because for any $\mathbf{T}$ with an infinite number of a's we can always take some $\mathbf{H}$ from which we remove a at some state, and still have an infinite number of a's in $\mathbf{H}$. Thus, for any total THT_ω-model $\langle \mathbf{T}, \mathbf{T}\rangle$ of $\Box\Diamond a$ there always exists some model $\langle \mathbf{H}, \mathbf{T}\rangle$ with strictly smaller $\mathbf{H} < \mathbf{T}$. Note that we can still specify infinite traces with an infinite number of occurrences of a, but at the price of *removing the truth minimization* for that atom. This can be done, for instance, by adding the excluded middle axiom (EM) for atom a. In this way, infinite traces satisfying $\Box\Diamond a \wedge \Box(a \vee \neg a)$ are those that contain an infinite number of a's. In fact, if we add the excluded middle axiom for all atoms, TEL collapses into LTL, as stated below.

Proposition 6. *Let Γ be a temporal theory over At and (EM) be the set of all excluded middle axioms for all atoms in At .*

Then, $TSM(\ \Gamma \cup (\mathrm{EM})\ ) = LTL(\Gamma)$. □

4 Computing Temporal Stable Models

Let us consider a more meaningful example, taking the Yale Shooting scenario [19] where we must shoot a loaded gun to kill a turkey. A possible encoding in TEL could be:

$$\Box(\mathit{loaded} \wedge \circ \mathit{shoot} \rightarrow \circ \mathit{dead}) \tag{6}$$
$$\Box(\mathit{loaded} \wedge \circ \mathit{shoot} \rightarrow \circ \mathit{unloaded}) \tag{7}$$
$$\Box(\mathit{load} \rightarrow \mathit{loaded}) \tag{8}$$
$$\Box(\mathit{dead} \rightarrow \circ \mathit{dead}) \tag{9}$$
$$\Box(\mathit{loaded} \wedge \neg\circ \mathit{unloaded} \rightarrow \circ \mathit{loaded}) \tag{10}$$
$$\Box(\mathit{unloaded} \wedge \neg\circ \mathit{loaded} \rightarrow \circ \mathit{unloaded}) \tag{11}$$

In this way, under TEL semantics, implication $\alpha \rightarrow \beta$ has a similar behaviour to a directional inference rule, normally reversed as $\beta \leftarrow \alpha$ or $\beta :- \alpha$ in logic programming notation. The last two rules, (10)-(11), encode the *inertia law* for fluents *loaded* and *unloaded*, respectively. Note the use of $\neg$ in these two rules: it actually corresponds to *default negation*, that is, $\neg\alpha$ is read as "there is no

evidence about α." For instance, (10) is read as "if the gun was loaded and we cannot prove that it will become unloaded then it stays loaded."

Computation of temporal stable models is a complex task. THT-satisfiability has been classified [8] as PSPACE-complete, that is, the same complexity as LTL-satisfiability, whereas TEL-satisfiability rises to EXPSPACE-completeness, as proved in [3]. In this way, we face a similar situation as in the non-temporal case where HT-satisfiability is NP-complete like SAT, whereas existence of equilibrium model (for arbitrary theories) is Σ_2^P-complete (like disjunctive ASP). There exist a pair of tools, `STeLP` [6] and `ABSTEM` [9], that allow computing (infinite) temporal stable models (represented as Büchi automata). These tools can be used to check verification properties that are usual in LTL, like the typical safety, liveness and fairness conditions, but in the context of temporal ASP. Moreover, they can also be applied for planning problems that involve an indeterminate or even infinite number of steps, such as the non-existence of a plan. In most practical problems, however, we are normally interested in finite traces. For that purpose, TEL$_f$ is implemented in the `telingo` system, extending the ASP system `clingo` to compute the temporal stable models of (non-ground) temporal logic programs. To this end, it extends the full-fledged input language of `clingo` with temporal operators and computes temporal models incrementally by multi-shot solving using a modular translation into ASP. `telingo` is freely available at `github`[1]. For instance, under `telingo` syntax, our theory (6)-(11) would be represented[2] as

```
#program dynamic.
dead :- shoot, 'loaded.
unloaded :- shoot, 'unloaded.
loaded :- load.
dead :- 'dead.
loaded :- 'loaded, not unloaded.
unloaded :- 'unloaded, not loaded.
```

The `telingo` input language actually allows the introduction of arbitrary LTL formulas in constraints or past formulas in the rule bodies (conditions). The syntax extends the full-fledged modeling language of `clingo` by the future and past temporal operators listed in the first and fourth row of Table 1. To support incremental ASP solving, `telingo` accepts a fragment of TEL$_f$ called *past-future rules* (see [11] for more details). A temporal formula is a past-future rule if it has form $Hd \leftarrow Bd$ where Bd and Hd are just temporal formulas with the following restrictions: Bd and Hd contain no implications (other than negations[3]), Bd contains no future operators, and Hd contains no past operators.

[1] `https://github.com/potassco/telingo`

[2] The left upper commas are read as *previously* and correspond to the past operator dual of *next* '$\bigcirc$'. The $\Box$ operator is implicit in all dynamic rules.

[3] Recall that $\neg\varphi \stackrel{\text{def}}{=} \varphi \rightarrow \bot$ in the logic of here-and-there and thus in TEL$_f$, too.

`&initial`	I	*initial*	`&final`	$\mathbb{F}$	*final*
`'p`	$\bullet p$	*previous*	`p'`	$\circ p$	*next*
`<`	$\bullet$	*previous*	`>`	$\circ$	*next*
`<?`	S	*since*	`>?`	$\mathbb{U}$	*until*
`<*`	T	*trigger*	`>*`	$\mathbb{R}$	*release*
`<?`	$\blacklozenge$	*eventually before*	`>?`	$\Diamond$	*eventually afterward*
`<*`	$\blacksquare$	*always before*	`>*`	$\Box$	*always afterward*
`<:`	$\widehat{\bullet}$	*weak previous*	`>:`	$\widehat{\circ}$	*weak next*

Table 1. Past and future temporal operators in `telingo` and TEL_f

An example of a past-future rule is, for instance, the formula

$$\Box(\mathit{shoot} \wedge \bullet\blacklozenge \mathit{shoot} \wedge \blacksquare \mathit{unloaded} \rightarrow \Diamond \mathit{fail}) \tag{12}$$

expressing the sentence: *"If we shoot twice with a gun that was never loaded, it will eventually fail."* The past-future fragment is not only quite expressive but also rather natural when using the causal reading of program rules by drawing upon the past in rule bodies and referring to the future in rule heads. Considering that, past-future rules also serve as the design guideline for `telingo`'s input language.

To this end, `telingo` allows for enclosing a nested temporal formula φ in an expression of the form `&tel`$\{\varphi\}$. Formulas like φ are formed via the temporal operators in Line 3 to 8 in Table 1 along with the Boolean operators `&`, `|`, `~` for conjunction, disjunction, and negation, respectively (thus avoiding nested implications). The underlying idea is to use the *smaller* symbol `<` as the basis of all past operators, and to combine it with a *question mark* `?` or a *Kleene star* `*` depending on whether the semantics of the respective operator relies on an existential or universal quantification over states. This is nicely exemplified by the always and eventually operators, represented by `<*` and `<?`. In fact, the symbols `<*` and `<?` are overloaded due to their usage as binary and unary operators. For a simple example, consider the formula $\bullet p \vee \blacklozenge r$ represented as '`&tel{< p | <? p}`'. Similarly, future operators are built with the *greater* symbol '`>`' as their basis. More generally, temporal expressions of the form `&tel`$\{\varphi\}$ are treated like atoms in `telingo`'s input language (and constitute theory atoms in `clingo` [**?**]); they are compiled away by `telingo`'s preprocessing that ultimately yields present-centered logic programs. In order to keep this translation simple, the current version of `telingo`, viz 2.1.1, restricts their occurrence in temporal rules $\mathit{Hd} \leftarrow \mathit{Bd}$ to being positive in Hd and preceded by one or two negations in their body Bd.[4] No restriction is imposed on their occurrences in integrity constraints. For example, the integrity constraint '$\mathit{shoot} \wedge \blacksquare \mathit{unloaded} \wedge \bullet\blacklozenge \mathit{shoot} \rightarrow \bot$' is expressible in several alternative ways.

[4] The extension to arbitrary occurrences is no hurdle and foreseen in future versions of `telingo`.

```
:- &tel { shoot & <* unloaded & < <? shoot }.
:- shoot,  &tel { <* unloaded & < <? shoot }.
:- shoot,  &tel { <* unloaded }, &tel { < <? shoot }.
```

Alternatively, present-centered logic programs can be written directly by using the alternative notation for the common one-step operators $\bullet$ and $\circ$. Here, a quote is used either at the beginning or the end of a predicate symbol to indicate that the literal at hand must be true in the previous or next state in the trace, respectively. For instance, $\bullet p(7)$ is represented by `'p(7)`, while $\circ q(X)$ is `q'(X)`. For convenience, `telingo` 2.1.1 allows for using $\circ$ in singleton rule heads;[5] as above, this is compiled away during preprocessing.

The distinction between different types of temporal rules is done in `telingo` via `clingo`'s #program directives [16], which allow us to partition programs into subprograms. More precisely, each rule in `telingo`'s input language is associated with a temporal rule r of form $Hd \leftarrow Bd$ and interpreted as r, $\widehat{\circ}\Box r$, or $\Box(\mathbb{F} \rightarrow r)$ depending on whether it occurs in the scope of a program declaration headed by initial , dynamic, or final , respectively. Additionally, `telingo` offers always for gathering rules preceded by $\Box$ (thus dropping $\widehat{\circ}$ from dynamic rules). A rule outside any such declaration is regarded to be in the scope of initial .

For illustration, we give in Listing 1.1 an exemplary `telingo` encoding of the *Fox, Goose and Beans Puzzle* available at `https://github.com/potassco/telingo/tree/master/examples/river-crossing`.

> *Once upon a time a farmer went to a market and purchased a fox, a goose, and a bag of beans. On his way home, the farmer came to the bank of a river and rented a boat. But crossing the river by boat, the farmer could carry only himself and a single one of his purchases: the fox, the goose, or the bag of beans. If left unattended together, the fox would eat the goose, or the goose would eat the beans. The farmer's challenge was to carry himself and his purchases to the far bank of the river, leaving each purchase intact. How did he do it?*
>
> (`https://en.wikipedia.org/wiki/Fox,_goose_and_bag_of_beans_puzzle`)

In Listing 1.1, lines 3-5 and 9-10 provide facts holding in all and the initial states, respectively; this is indicated by the program directives headed by `always` and `initial`. The dynamic rules in lines 14-22 describe the transition function. The `farmer` moves at each time step (Line 14), and may take an item or not (Line 15). Line 17 describes the effect of action `move/1`, Line 18 its precondition, and Line 20 the law of inertia. The second part of the `always` rules give state constraints in Line 24 and 25. The `final` rule in Line 29 gives the goal condition.

All in all, we obtain two shortest plans consisting of eight states in about 20 ms. Restricted to the move predicate, `telingo` reports the following solutions:

[5] As above, the extension to disjunctions is no principal hurdle and foreseen in future versions of `telingo`; currently they must be expressed by using `&tel`.

Listing 1.1. `telingo` encoding for the Fox, Goose and Beans Puzzle

```
#program always.

item(fox;beans;goose).
route(river_bank,far_bank). route(far_bank,river_bank).
eats(fox,goose). eats(goose,beans).

#program initial.

at(farmer,river_bank).
at(X,river_bank) :- item(X).

#program dynamic.

move(farmer).
0 { move(X) : item(X) } 1.

at(X,B) :- 'at(X,A), move(X), route(A,B).
:- move(X), item(X), 'at(farmer,A), not 'at(X,A).

at(X,A) :- 'at(X,A), not move(X).

#program always.

:- at(X,A), at(X,B), A<B.
:- eats(X,Y), at(X,A), at(Y,A), not at(farmer,A).

#program final.

:- at(X,river_bank).

#show move/1.
#show at/2.
```

Time	Solution 1		Solution 2	
1				
2	move(farmer)	move(goose)	move(farmer)	move(goose)
3	move(farmer)		move(farmer)	
4	move(beans)	move(farmer)	move(farmer)	move(fox)
5	move(farmer)	move(goose)	move(farmer)	move(goose)
6	move(farmer)	move(fox)	move(beans)	move(farmer)
7	move(farmer)		move(farmer)	
8	move(farmer)	move(goose)	move(farmer)	move(goose)

We have chosen this example since it was also used by [6] to illustrate the working of STeLP, a tool for temporal answer set programming with TEL$_\omega$. We note that STeLP and telingo differ syntactically in describing transitions by using next or previous operators, respectively. Since telingo extends clingo's input language, it offers a richer input language, as witnessed by the cardinality constraints in Line 15 in Listing 1.1. Finally, STeLP uses a model checker and outputs an automaton capturing all infinite traces while telingo returns finite traces corresponding to plans.

As a second example, consider the following problem[6] proposed by Professor J. Moore from the University of Texas at Austin and submitted to the Texas Action Group (TAG) discussion group.

> *Consider two processes, A and B, each of which is reading and writing a shared variable C. Each process is in an infinite loop, repeatedly executing: C = C + C; By this we mean "read the value of C, read the value of C again, add the two results and store the sum in C." The two reads store the values in "local registers" of the process. Reads and writes are atomic but there is no synchronization between the two processes. The initial value of C is 1. Problem: Given an arbitrary positive integer n is there an execution that assigns C the value n?*

A first possible encoding of this problem in telingo could look simply be the one shown in Listing 1.2. In this encoding, action `fetch(P)` reflects the fact that the CPU has non-deterministically decided to execute the next instruction from process P, encoding the process interleaving in that way. This non-deterministic choice is encoded in lines 23-26. The most significative feature of this encoding is that we keep an auxiliary fluent `i(P)` to stand for the current *instruction pointer*of each process P. An interesting observation is that, once we allow temporal expressions in the rule bodies and constraints, we can sometimes replace auxiliary fluents in favour of temporal queries about the past execution. In our example, this leads to a second encoding shown in Listing 1.3. In this case, we have removed the fluent capturing the instruction pointer and replaced it by the constraints in lines 33-35. Lines 33-34 mean that if we fetch instruction I for a process P already fetched in the past, then the last instruction fetched for P (whenever this is located in the past) must be I+2 modulo 3 (that is, the previous one in the cyclic order of instructions 0,1,2,0,...). Line 35 forces that the first instruction to be executed by any process P is 0: we cannot fetch instructions 1 or 2 if process P has not been already fetched.

5 Conclusions

These recent results open several interesting topics for future study. First, it would be interesting to adapt existing model checking techniques (based on automata construction) for temporal logics to solve the problem of existence

[6] https://www.cs.utexas.edu/users/vl/tag/jmoore_discussion

of temporal stable models. This was done for infinite traces in [8, 6], but no similar method has been implemented for finite traces on TEL_f. The importance of having an efficient implementation of such a method is that it would allow deciding non-existence of a plan in a given planning problem, something not possible by current incremental solving techniques. Another interesting topic is the optimization of grounding in temporal ASP specifications as those handled by `telingo`. The current grounding of `telingo` is inherited from incremental solving in `clingo` and does not exploit the semantics of temporal expressions that are available now in the input language. Finally, we envisage to extend the `telingo` system with features of DEL (an extension to cope with dynamic logic operators) in order to obtain a powerful system for representing and reasoning about dynamic domains, not only providing an effective implementation of TEL and DEL but, furthermore, a platform for action and control languages.

Acknowledgements This document contains partial reproductions of joint work with my coauthors Felicidad Aguado, Martín Diéguez, Jorge Fandinno, Roland Kaminski, Philip Morkisch, David Pearce, Gilberto Pérez, Torsten Schaub, Anna Schuhmann, Agustín Valverde and Concepción Vidal, all of them involved in the development of Equilibrium Logic, particularly in its temporal extension and its application to practical ASP solving. I wish to thank all of them for the fruitful collaboration and friendship along these years and extend this recognition to other colleagues that actively participate in the discussions or development of linear-temporal ASP like François Laferriere, Susana Hahn, Etienne Tignon and Javier Romero from the Potassco group at Potsdam and Philip Balbiani, Andreas Herzig and Luis Fariñas del Cerro from the IRIT at Toulouse, among others.

References

1. Aguado, F., Cabalar, P., Diéguez, M., Pérez, G., Schaub, T., Schuhmann, A., Vidal, C.: Linear-time temporal answer set programming. Theory and Practice of Logic Programming (submitted) (2021)
2. Aguado, F., Cabalar, P., Diéguez, M., Pérez, G., Vidal, C.: Temporal equilibrium logic: a survey. Journal of Applied Non-Classical Logics **23**(1-2), 2–24 (2013)
3. Bozzelli, L., Pearce, D.: On the complexity of temporal equilibrium logic. In: Proceedings of the 30th Annual ACM/IEEE Symposium of Logic in Computer Science (LICS'15). Kyoto, Japan (2015), (to appear)
4. Brewka, G., Eiter, T., Truszczyński, M.: Answer set programming at a glance. Communications of the ACM **54**(12), 92–103 (2011)
5. Bylander, T.: The computational complexity of propositional strips planning. Artificial Intelligence **69**(1), 165 – 204 (1994)
6. Cabalar, P., Diéguez, M.: STELP - a tool for temporal answer set programming. In: LPNMR'11. Lecture Notes in Computer Science, vol. 6645, pp. 370–375 (2011)
7. Cabalar, P., Perez, G.: Temporal Equilibrium Logic: A First Approach. In: Proceedings of the 11^{th} International Conference on Computer Aided Systems Theory (EUROCAST'07). p. 241248 (2007)
8. Cabalar, P., Demri, S.: Automata-based computation of temporal equilibrium models. In: 21st International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'11) (2011)

9. Cabalar, P., Diéguez, M.: Strong equivalence of non-monotonic temporal theories. In: Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR'14). Vienna, Austria (2014)
10. Cabalar, P., Kaminski, R., Morkisch, P., Schaub, T.: telingo = ASP + time. In: Balduccini, M., Lierler, Y., Woltran, S. (eds.) Proc. of the 15th Intl. Conf. on Logic Programming and Nonmonotonic Reasoning, LPNMR 2019, Philadelphia, PA, USA. Lecture Notes in Computer Science, vol. 11481, pp. 256–269. Springer (2019)
11. Cabalar, P., Kaminski, R., Schaub, T., Schuhmann, A.: Temporal answer set programming on finite traces. Theory and Practice of Logic Programming **18**(3-4), 406–420 (2018)
12. Cabalar, P., Pearce, D., Valverde, A.: Answer set programming from a logical point of view. Künstliche Intelligenz **32**(2-3), 109–118 (2018)
13. De Giacomo, G., Vardi, M.: Linear temporal logic and linear dynamic logic on finite traces. In: Rossi, F. (ed.) Proceedings of the Twenty-third International Joint Conference on Artificial Intelligence (IJCAI'13). pp. 854–860. IJCAI/AAAI Press (2013)
14. van Emden, M.H., Kowalski, R.A.: The semantics of predicate logic as a programming language. Journal of the ACM **23**, 733–742 (1976)
15. Fages, F.: Consistency of Clark's completion and the existence of stable models. Journal of Methods of Logic in Computer Science **1**, 51–60 (1994)
16. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Multi-shot ASP solving with clingo. Theory and Practice of Logic Programming **19**(1), 27–82 (2019), `http://arxiv.org/abs/1705.09811`
17. Gelfond, M., Lifschitz, V.: The Stable Model Semantics For Logic Programming. In: Proc. of the 5th International Conference on Logic Programming (ICLP'88). p. 10701080. Seattle, Washington (1988)
18. Gelfond, M., Lifschitz, V.: Representing action and change by logic programs. Journal of Logic Programming **17**(2/3&4), 301–321 (1993)
19. Hanks, S., McDermott, D.V.: Nonmonotonic logic and temporal projection. Artificial Intelligence **33**(3), 379–412 (1987)
20. Heyting, A.: Die formalen Regeln der intuitionistischen Logik. Sitzungsberichte der Preussischen Akademie der Wissenschaften. Physikalisch-mathematische Klasse (1930)
21. Pearce, D.: A New Logical Characterisation of Stable Models and Answer Sets. In: Proc. of Non-Monotonic Extensions of Logic Programming (NMELP'96). pp. 57–70. Bad Honnef, Germany (1996)
22. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science. pp. 46–57. IEEE Computer Society Press (1977)

Listing 1.2. `telingo` basic encoding for Moore's problem

```
#const n=19.

#program initial.

process(a;b).
local(0;1).
instruction(0..2).

% Each process P executes:
% 0   assign c to r(P)
% 1   add c to r(P)
% 2   assing r(P) to C
%
% i(P)=I points to the next instruction I of process P to execute

holds(i(P),0) :- process(P).
holds(c,1).
holds(r(P),0) :- process(P).

#program dynamic.

%1 {fetch(P): _process(P)} 1.
{fetch(P,I):_instruction(I)} 1 :- _process(P).
fetch(P) :- fetch(P,I).
:- fetch(P,I), not _local(I), fetch(Q), P!=Q.
:- #count{P:fetch(P)}=0.

change(i(P),(I+1)\3) :- fetch(P), 'holds(i(P),I).
change(r(P),C  )     :- fetch(P), 'holds(i(P),0), 'holds(c,C).
change(r(P),R+C)     :- fetch(P), 'holds(i(P),1), 'holds(c,C),
                        'holds(r(P),R), R+C <= n.
change(r(P),n+1)     :- fetch(P), 'holds(i(P),1), 'holds(c,C),
                        'holds(r(P),R), R+C >  n.
change(c   ,R  )     :- fetch(P), 'holds(i(P),2), 'holds(r(P),R).

holds(F,V) :- change(F,V).
holds(F,V) :- 'holds(F,V), not change(F,_).

#program final.
:- not _testing, not holds(c,n).

#show fetch/1.
#show holds/2.
```

Listing 1.3. telingo second encoding for Moore's problem

```
#const n=23.

#program initial.

process(a;b).
local(0;1).
instruction(0..2).

% Each process P executes:
% 0   assign c to r(P)
% 1   add c to r(P)
% 2   assing r(P) to C

holds(c,1).
holds(r(P),0) :- process(P).

#program dynamic.

{fetch(P,I): _instruction(I)} 1 :- _process(P).
fetch(P) :- fetch(P,I).
:- fetch(P,I), not _local(I), fetch(Q), P!=Q.
:- #count{P:fetch(P)}=0.


change(r(P),C  ) :- fetch(P,0), 'holds(c,C).
change(r(P),R+C) :- fetch(P,1), 'holds(c,C), 'holds(r(P),R), R+C<=n.
change(r(P),n+1) :- fetch(P,1), 'holds(c,C), 'holds(r(P),R), R+C>n.
change(c   ,R  ) :- fetch(P,2), 'holds(r(P),R).

holds(F,V) :- change(F,V).
holds(F,V) :- 'holds(F,V), not change(F,_).

:- fetch(P,I), &tel{< <? fetch(P)},
   not &tel { < (~fetch(P) <? fetch(P,I')) : I'=(I+2)\3 }.
:- fetch(P,I), not &tel{< <? fetch(P)}, I!=0.

#program final.
:- not _testing, not holds(c,n).

#show fetch/2.
#show holds/2.
```