telingo = ASP + Time

Pedro Cabalar$^{1[0000-0001-7440-0953]}$, Roland Kaminski$^{2[0000-0002-1361-6045]}$, Philip Morkisch$^{2[0000-0002-3915-2808]}$, and Torsten Schaub$^{2\star[0000-0002-7456-041X]}$

^{1} University of Corunna, Spain
^{2} University of Potsdam, Germany

Abstract. We describe *telingo*, an extension of the ASP system *clingo* with temporal operators over finite linear time and provide insights into its implementation. *telingo* takes temporal logic programs as input whose rules contain only future and present operators in their heads and past and present operators in their bodies. Moreover, *telingo* extends the grammar of *clingo*'s input language with a variety of temporal operators that can even be used to represent nested temporal formulas. By using *clingo*'s interface for manipulating the abstract syntax tree of non-ground programs, temporal logic programs are transformed into regular ones before grounding. The resulting regular logic program is then solved incrementally by using *clingo*'s multi-shot interface. Notably, this involves the consecutive unfolding of future temporal operators that is accomplished via external atoms. Finally, we provide an empirical evaluation contrasting standard incremental ASP programs with their temporal counterparts in *telingo*'s input language.

1 Introduction

Answer Set Programming (ASP [15]) has become a popular approach to solving knowledge-intense combinatorial search problems due to its performant solving engines and expressive modeling language. However, both are mainly geared towards static domains and lack native support for handling dynamic applications. This shortcoming was addressed over the last decade by creating a temporal extension of ASP based on Linear Temporal Logic (LTL [17]) and referred to as Temporal Equilibrium Logic (TEL [5, 1]). Recently, this was distilled into a computationally more feasible version based on finite linear time. The resulting logic, TEL_f [4] has meanwhile led to the temporal ASP system *telingo* [4], which we describe in this system description. *telingo* extends the full-fledged modeling language of the ASP system *clingo* by future and past temporal operators and solves the corresponding temporal logic programs incrementally by means of *clingo*'s multi-shot solving interface. Hence, we also provide insights into how *clingo*'s infrastructure can be used to implement more complex ASP languages.

2 Temporal equilibrium logic over finite traces

The semantics of TEL_f rests upon finite traces (or sequences) of equilibrium models (cf. [4]), just as LTL_f rests upon finite traces of regular models [6]. In fact, LTL_f is

* Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

obtained by adding the law of the excluded middle for each propositional atom and each time point, or in terms of ASP, by adding a corresponding choice rule (see below). Hence, *telingo* can be used just as well for computational tasks in LTL_f.

TEL_f extends the language of propositional logic by the future and past temporal operators listed in the second and fifth row of Table 1. The first line gives the two nullary

`&initial`	$\mathbf{I}$	*initial*	`&final`	$\mathbb{F}$	*final*
`'p`	$\bullet p$	*previous*	`p'`	$\circ p$	*next*
`<`	$\bullet$	*previous*	`>`	$\circ$	*next*
`<?`	$\mathbf{S}$	*since*	`>?`	$\mathbb{U}$	*until*
`<*`	$\mathbf{T}$	*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

operators $\mathbf{I}$ and $\mathbb{F}$ that hold exclusively at the initial and final state of a trace, respectively. The common one-step operators $\bullet$ and $\circ$ allow us to test whether a proposition holds in the previous or next state in a trace, respectively. Their weak versions are defined as $\widehat{\bullet}\varphi \overset{def}{=} \bullet\varphi \vee \mathbf{I}$ and $\widehat{\circ}\varphi \overset{def}{=} \circ\varphi \vee \mathbb{F}$, respectively. The unary operators $\blacklozenge$ and $\blacksquare$ allow us to refer to one or all states in the past, respectively, while their counterparts $\Diamond$ and $\Box$ relate to the future. Common to all operators, a bold version indicates a past operator, while an outlined one refers to the future. As a simple example, the proposition $\bullet\Box p$ requires that p must be true in all states of a trace starting from the state preceding the one at hand. For another example, consider the formula

$$\Box(\mathit{shoot} \wedge \bullet\blacklozenge\mathit{shoot} \wedge \blacksquare\mathit{unloaded} \rightarrow \Diamond\mathit{fail}) \tag{1}$$

expressing the sentence: *"If we shoot twice with a gun that was never loaded, it will eventually fail."* Finally, an atom p is put under the semantics of LTL_f by adding $\Box(p \vee \neg p)$. For the binary operators $\mathbf{S}$, $\mathbf{T}$, $\mathbb{U}$, and $\mathbb{R}$ along with more details and illustration regarding the temporal language and its semantics the interested reader is referred to [1, 4].

Any temporal formula can be translated into a (strongly equivalent) *temporal logic program*. Given an alphabet $\mathcal{A}$, such programs consist of three types of *temporal rules*

– *initial rules* of form $B \rightarrow A$
– *dynamic rules* of form $\widehat{\circ}\Box(B \rightarrow A)$
– *final rules* of form $\Box(\,\mathbb{F} \rightarrow (B \rightarrow A)\,)$

where $B = b_1 \wedge \cdots \wedge b_n$ with $n \geq 0$, $A = a_1 \vee \cdots \vee a_m$ with $m \geq 0$ and the b_i and a_j are *temporal literals* as in $\{a, \neg a, \bullet a, \neg\bullet a \mid a \in \mathcal{A}\}$ for dynamic rules, and regular literals $\{a, \neg a \mid a \in \mathcal{A}\}$ for initial and final rules.

As their names suggest, initial and final rules impose conditions on the first and last state of a trace, respectively. The former can also be expressed in analogy to the latter as $\square(\mathbf{I} \rightarrow (B \rightarrow A))$. A temporal program consisting of initial rules only amounts to a regular logic program. Dynamic rules capture transitions among states. To this end, they comprise regular and temporal literals that may refer to a preceding state via the previous operator $\bullet$. To avoid referring to states beyond the initial and final state, dynamic rules are preceded with the weak next operator $\widehat{\circ}$ operator.

A temporal logic program can be converted into a regular one by adorning literals with explicit timestamps (cf. [14]). For this, let $\mathcal{A}_k = \{a_k \mid a \in \mathcal{A}\}$ be a time stamped copy of alphabet $\mathcal{A}$ for each time point k. We outline below the module-based translation introduced in [4] since it accounts for *telingo*'s incremental approach to computing traces: A module $\mathbb{P}$ is a triple (P, I, O) consisting of a logic program P over alphabet $\mathcal{A}_P$ and sets I and O of input and output atoms such that (i) $I \cap O = \emptyset$, (ii) $\mathcal{A}_P \subseteq I \cup O$, and (iii) $H(P) \subseteq O$, where $H(P)$ gives all atoms occurring in rule heads in P (cf. [16]). Whenever clear from context, we associate $\mathbb{P}$ with (P, I, O). In our setting, a set X of atoms is a stable model of $\mathbb{P}$, if X is a stable model of logic program P.[3] Two modules $\mathbb{P}_1$ and $\mathbb{P}_2$ are *compositional*, if $O_1 \cap O_2 = \emptyset$ and $O_1 \cap C = \emptyset$ or $O_2 \cap C = \emptyset$ for every strongly connected component C of the positive dependency graph of the logic program $P_1 \cup P_2$. In other words, all rules defining an atom must belong to the same module, and no positive recursion is allowed among modules. Whenever $\mathbb{P}_1$ and $\mathbb{P}_2$ are compositional, their *join* is defined as the module $\mathbb{P}_1 \sqcup \mathbb{P}_2 = (P_1 \cup P_2, (I_1 \setminus O_2) \cup (I_2 \setminus O_1), O_1 \cup O_2)$. The module theorem [16] ensures that compatible stable models of $\mathbb{P}_1$ and $\mathbb{P}_2$ can be mapped to one of $\mathbb{P}_1 \sqcup \mathbb{P}_2$, and vice versa.

Given this, the translation τ at time point k is defined for temporal literals as

$$\begin{array}{lll} \tau_k(a) \overset{def}{=} a_k & \tau_k(\neg a) \overset{def}{=} \neg a_k & \text{for } a \in \mathcal{A} \\ \tau_k(\bullet a) \overset{def}{=} a_{k-1} & \tau_k(\neg\bullet a) \overset{def}{=} \neg a_{k-1} & \text{for } a \in \mathcal{A} \end{array}$$

and for temporal rules r in a temporal logic program P partitioned into its initial, $I(P)$, dynamic, $D(P)$, and final rules, $F(P)$, as

$$\begin{array}{ll} \tau_k(r) \overset{def}{=} \tau_k(a_1) \vee \cdots \vee \tau_k(a_m) \leftarrow \tau_k(b_1) \wedge \cdots \wedge \tau_k(b_n) & \text{if } r \in I(P) \cup D(P) \\ \tau_k(r) \overset{def}{=} \tau_k(a_1) \vee \cdots \vee \tau_k(a_m) \leftarrow \tau_k(b_1) \wedge \cdots \wedge \tau_k(b_n) \wedge \neg q_{k+1} & \text{if } r \in F(P) \end{array}$$

for a new atom $q \notin \mathcal{A}$. The modules $\mathbb{P}_k$ corresponding to a temporal logic program P over $\mathcal{A}$ at time point k are then defined as

$$\mathbb{P}_0 \overset{def}{=} (P_0, \{q_1\}, \mathcal{A}_0) \qquad \mathbb{P}_k \overset{def}{=} (P_k, \mathcal{A}_{k-1} \cup \{q_{k+1}\}, \mathcal{A}_k \cup \{q_k\}) \quad \text{for } k > 0$$

where

$$P_0 \overset{def}{=} \{\tau_0(r) \mid r \in I(P)\} \cup \{\tau_0(r) \mid r \in F(P)\} \tag{2}$$

$$P_k \overset{def}{=} \{\tau_k(r) \mid r \in D(P)\} \cup \{\tau_k(r) \mid r \in F(P)\} \cup \{q_k \leftarrow\} \tag{3}$$

The idea is to associate the rules at each time point with a module and to successively add modules corresponding to increasing time points (while leaving all previous modules

[3] Note that the default value assigned to input atoms is *false* in multi-shot solving [10]; this differs from the original definition [16] where a choice rule is used.

unchanged). A stable model obtained after k compositions then corresponds to a trace of length k.

To ensure the compositionality of modules, dynamic rules are restricted to heads of regular literals; such rules are called *present-centered* [4]. This restriction warrants that modules only incorporate atoms from previous time points, as reflected by $\mathcal{A}_{k-1}$ in the input of $\mathbb{P}_k$, and thus that no positive cycles can occur across modules.

The exception are auxiliary atoms like q_{k+1} that belong to the input of each $\mathbb{P}_k$ for $k > 0$ but only get defined in the next module $\mathbb{P}_{k+1}$. The goal of introducing atoms like q_{k+1} in the translation of final rules $r \in F(P)$ is to deactivate their image $\tau_k(r)$ whenever k is incremented. More precisely, the idea is to let atom q_{k+1} be false at each horizon k (by declaring it as a yet undefined input atom), while all previous atoms $q_1, \dots, q_k$ are set to true via the facts added in $P_1, \dots, P_k$, respectively. In this way, for $r \in F(P)$ only $\tau_k(r)$ is potentially applicable at time point k, while all rules $\tau_i(r)$ are inapplicable for earlier time points $i = 1..k{-}1$.

3 The *telingo* language

telingo 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.

Although *telingo*'s inner workings rely on present-centered temporal logic programs (to support incremental ASP solving), it offers a more general input language. This is because the fragment of *past-future rules* is reducible to present-centered programs [4]. A temporal formula is a past-future rule if it has form $A \leftarrow B$ where B and A are just temporal formulas with the following restrictions: B and A contain no implications (other than negations[4]), B contains no future operators, and A contains no past operators. An example of a past-future rule is (1). This 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* [9]); they are compiled away by *telingo*'s preprocessing that ultimately yields present-centered logic programs. In order

[4] Recall that $\neg\varphi \overset{def}{=} \varphi \rightarrow \bot$ in the logic of here-and-there and thus in TEL_f, too.

to keep this translation simple, the current version of *telingo*, viz 1.0, restricts their occurrence in temporal rules $A \leftarrow B$ to being positive in A and preceded by one or two negations in their body B.[5] 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.

```
:- &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* 1.0 allows for using $\circ$ in singleton rule heads;[6] 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 [10], 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 $A \leftarrow B$ 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 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, 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.

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

[6] 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`.

```
#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.
```

Listing 1. *telingo* encoding for the Fox, Goose and Beans Puzzle

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:

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 [3] to illustrate the working of *stelp*, a tool for temporal answer set programming with TEL_ω. 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. Finally, *stelp* uses a model checker and outputs an automaton capturing all infinite traces while *telingo* returns finite traces corresponding to plans.

4 The *telingo* system

The implementation of *telingo* draws heavily on the functionality provided by *clingo*'s application programming interface (API[7]). This is also why *telingo* allows us to extend the full-fledged modeling and solving capabilities of *clingo*.

We outline *telingo*'s operation below by following its workflow.

4.1 Parsing temporal logic programs

All of the temporal language additions are designed to use available syntax features, so that *clingo*'s (or better *gringo*'s [11]) parser can be used as is. Atoms like `&initial`, `&final`, and `&tel`, as well as the temporal operators in the first and fourth column of Table 1 rely on *clingo*'s theory language capacities that allow for defining customized syntactic expressions by supplementing a dedicated (theory) grammar (cf. [9]). Also, *clingo* tolerates quotes in predicate names. Finally, *telingo* uses *clingo*'s `#program` directive [10] for partitioning temporal logic programs into their four types of rules.

4.2 Translating temporal logic programs into regular ones

The translation of temporal logic programs into regular ones relies on the processing of the temporal adornments described in Section 4.1. This information is used for generating an (incremental) logic program, as described in Section 2. In practice, the resulting program is equipped with program directives that allow *clingo* to use its multi-shot solving capabilities (cf. [13, 10]) for solving the program incrementally. The actual translation is accomplished by means of the functionalities of *clingo*'s API for manipulating the abstract syntax tree of a logic program. That is, the list of rules is extracted, rewritten, and finally passed back to *clingo*.

The most intriguing part in this process is the (incremental) rewriting of future-oriented operators in heads of past-future rules. In fact, the restriction of having future operators occur in rule heads only and past operators occur in rule bodies results in a normal form where all future operators occur negatively in rule bodies and rule heads do not contain temporal operators anymore. In general, this normal form creates a temporal program with an infinite number of rules but only a finite number of them are required for a fixed horizon. The translation rests on the idea that for past-future rules there can be no positive cycles involving an atom from the current step and an atom from a future step. This allows us to shift rule heads and bring a program in the above normal form. The formal elaboration of this translation is detailed in a companion paper, and we focus below on an example-driven presentation.

[7] `https://potassco.org/clingo`

Let us begin by illustrating the elimination of negative occurrence of future operators in rule bodies. As just mentioned, they appear during *telingo*'s translation in an intermediate step but can be turned back into present-centered temporal logic programs.[8] For example, consider an occurrence of $\neg\circ\Diamond a$ (viz. '`not &tel { > >? a }`') in a rule body, since this pops up below again. Each such negative occurrence of $\circ\Diamond a$ is replaced by an auxiliary atom $\ell_{\circ\Diamond a}$:

$$\Box(A \leftarrow \neg\circ\Diamond a \wedge B) \quad \mapsto \quad \Box(A \leftarrow \neg\ell_{\circ\Diamond a} \wedge B)$$

Since the occurrence of $\circ\Diamond a$ is negative, TEL allows us to treat it as in classical (linear time) logic, namely by starting from $\Box(\ell_{\circ\Diamond a} \leftrightarrow \circ\Diamond a)$, we get $\Box(\ell_{\circ\Diamond a} \leftrightarrow \circ a \vee \circ\circ\Diamond a)$,[9] which we decompose into three integrity constraints in the standard way:

$$\begin{aligned}
&\Box(\ell_{\circ\Diamond a} \vee \neg\ell_{\circ\Diamond a}) \\
&\Box(\bot \leftarrow \ell_{\circ\Diamond a} \wedge \neg\circ a \wedge \neg\circ\circ\Diamond a) && (4)\\
&\Box(\bot \leftarrow \neg\ell_{\circ\Diamond a} \wedge \circ a) \\
&\Box(\bot \leftarrow \neg\ell_{\circ\Diamond a} \wedge \circ\circ\Diamond a) && (5)
\end{aligned}$$

While the first rule makes us choose the truth value of $\ell_{\circ\Diamond a}$, the last three rules result from rewriting the above equivalence (into two classical implications).

Finally, replacing in (4) and (5) the remaining occurrences of $\circ\Diamond a$ by $\ell_{\circ\Diamond a}$ and time shifting the inner part backwards by one and the outer one forward again by prepending $\widehat{\circ}$ results in the following set of present-centered rules

$$\begin{aligned}
&\Box(\ell_{\circ\Diamond a} \vee \neg\ell_{\circ\Diamond a}) \\
&\widehat{\circ}\Box(\bot \leftarrow \bullet\ell_{\circ\Diamond a} \wedge \neg a \wedge \neg\ell_{\circ\Diamond a}) \\
&\widehat{\circ}\Box(\bot \leftarrow \neg\bullet\ell_{\circ\Diamond a} \wedge a) \\
&\widehat{\circ}\Box(\bot \leftarrow \neg\bullet\ell_{\circ\Diamond a} \wedge \ell_{\circ\Diamond a}) \\
&\Box((\bot \leftarrow \ell_{\circ\Diamond a}) \leftarrow \mathbb{F}) && (6)
\end{aligned}$$

all of which are now ready to be compiled into regular rules with the translation given in Section 2. The application of the (weak) next operator shifts the temporal context of the actual rules one step ahead; the usage of the weak version $\widehat{\circ}$ makes sure that they are not falsified at the end of the trace. The final rule in (6) is added to ensure that $\ell_{\circ\Diamond a}$ is false in the final state. All in all, we get a translation linear in the size of the original literal.

Now, to illustrate the actual rewriting of future-oriented operators in rule heads, let us start with a simple past-future temporal rule

$$\Box(a \vee \circ b \leftarrow \ell) \qquad (7)$$

where ℓ can be thought of as an auxiliary atom, representing the original body. $a \vee \circ b$ means that a is true now or b is true at the next point in time.

[8] This is also why this extension to the past-future format is tolerated in *telingo*'s input language.
[9] $\circ\Diamond a \leftrightarrow \circ a \vee \circ\circ\Diamond a$ is valid in TEL.

The translation consists of three parts. First, we time-shift a rule for all possible time points, in which an atom, viz. a or b, can be made true. For the above rule, there are only two relevant rules:

$$\begin{aligned}\square(a \vee \circ b \leftarrow \ell)\\ \widehat{\circ}\square(\bullet a \vee b \leftarrow \bullet\ell)\end{aligned}$$

Until this point, both rules together are strongly equivalent to the original one in (7). Note that $\bullet a \vee b$ means (outside of any temporal context) that a is true at the previous point in time or b is true now.

Then, in the resulting rules, we double negate each outermost next and previous operator in the rule head. For our example, this results in:

$$\begin{aligned}\square(a \vee \neg\neg\circ b \leftarrow \ell)\\ \widehat{\circ}\square(\neg\neg\bullet a \vee b \leftarrow \bullet\ell)\end{aligned}$$

Here, we loose strong equivalence but the past-future condition guarantees that the solutions of the obtained programs are the same.

Finally, we can unfold the formulas in the usual way. For our example, this is:

$$\begin{aligned}\square(a \leftarrow \neg\circ b \wedge \ell)\\ \widehat{\circ}\square(b \leftarrow \neg\bullet a \wedge \bullet\ell)\end{aligned}$$

This is strongly equivalent to the rules obtained in the previous step of the translation. Once the negative occurrence of $\circ b$ is eliminated from the first rule (as shown above), we get a set of present-centered dynamic rules being equivalent to the one in (7).

Finally, let us consider the treatment of an inductive operator, and have a look at the eventually operator in the head of the following rule:

$$\square(\Diamond a \leftarrow \ell) \tag{8}$$

$\Diamond a$ means that a is true now or at some point in the future; its unfolding relies on the temporal law $\Diamond a \leftrightarrow a \vee \circ\Diamond a$.

By letting $\bullet^0\varphi = \varphi$ and $\bullet^i\varphi = \bullet\bullet^{i-1}\varphi$ for $i > 0$, we obtain in step one:

$$\begin{aligned}\square(a \vee \circ\Diamond a \leftarrow \bullet^0\ell)\\ \square(\bullet^1 a \vee a \vee \circ\Diamond a \leftarrow \bullet^1\ell)\\ \vdots\\ \square(\bullet^i a \vee \cdots \vee \bullet^1 a \vee a \vee \circ\Diamond a \leftarrow \bullet^i\ell)\end{aligned}$$

Taken together, these rules are equivalent to the rule in (8) but differ in the number of applications of the law $\Diamond a \leftrightarrow a \vee \circ\Diamond a$.[10]

[10] Unlike in the example above, we do not obtain strongly equivalent rules because we do not introduce weak next operators. This is safe in this context because the literal $\bullet^i\ell$ does not apply for horizons smaller i.

For each $i \geq 0$, we can then add the double negations as in the example above:

$$\Box(\neg\neg\bullet^i a \vee \cdots \vee \neg\neg\bullet^1 a \vee a \vee \neg\neg\circ\Diamond a \leftarrow \bullet^i \ell)$$

And finally, we can shift the double negated literals into the rule body:

$$\Box(a \leftarrow \neg\bullet^i a \wedge \cdots \wedge \neg\bullet^1 a \wedge \neg\circ\Diamond a \wedge \bullet^i \ell)$$

Once all negative occurrences of $\circ\Diamond a$ are eliminated (as shown above), we get once more a linear number of present-centered dynamic rules (of successively increasing size) being equivalent to the one in (8). In this case, we thus get a translation of quadratic size.

In general, the unfolding of future formulas may result in an exponential translation whereas the one for past formulas is linear in size. Currently, *telingo* unfolds without introducing shortcuts. We might be able to use a full Tseitin-style translation introducing auxiliary atoms to keep the translation compact, along with a good strategy guaranteeing compactness, which might be more difficult in the presence of inductive operators.

4.3 Solving regular logic programs incrementally

The above translation results in two (non-ground) regular logic programs corresponding to P_0 and P_k in (2) and (3), respectively. A control loop, similar to the one in [13], starts with P_0 and successively adds P_k for increasing k, grounds it, and solves the accumulated program until a stop criterion is met. In other words, *telingo* computes the stable models of $P_0 \cup \bigcup_{k\geq 0} P_k$ for $k \geq 0$. This process is controlled by three options: `--imin` and `--imax`, the minimum and maximum number of solving steps, respectively, and `--istop`, the stop criterion, which defaults to `sat` and offers alternatives `unsat` and `unknown`.

An interesting detail concerns the treatment of the final rule in Line 29 of Listing 1, viz. '`:- at(X,river_bank).`' standing for $\Box(\mathbb{F} \rightarrow \neg\mathtt{at}(\mathtt{X},\mathtt{river\_bank}))$. As described in Section 2, final rules are equipped with a special-purpose literal, $\neg q_{k+1}$, during their translation into regular rules in order to control their range of applicability in view of increasing k. In terms of module theory, q_{k+1} is an input atom, and they are accounted for by `#external` declarations in *clingo*. In our example, the recurrent test of '`:- at(X,river_bank).`' at the final time point gives rise to the program:

```
1  #external q(k+1). [false]
2  :- at(X,river_bank,k), not q(k+1).
```

The time parameter `k` is handled by the aforementioned control loop through *clingo*'s API. The declaration of `q(k+1)` as an external exempts it from simplification and allows for assigning truth values via the API. The trailing `[false]` gives the initial truth value.[11] For brevity, we refrain from duplicating the rule for all instantiations of `X`. For $\mathtt{k} = 0$, we thus get '`:- at(X,river_bank,0), not q(1).`' along with `q(1)` being `false`. Hence, the integrity constraint amounts to requiring that `at(X,river_bank)` is false for all instantiations of `X` at time point `0`.

For $\mathtt{k} = 1$, we have two instances of Line 2:

[11] This feature is introduced with *clingo* 5.4.

```
:- at(X,river_bank,0), not q(1).
:- at(X,river_bank,1), not q(2).
```

However, while as before `q(2)` is set to `false` by its declaration as an external (cf. Line 1 above), the control loop changes the truth of `q(1)` to `true`. As a result, the first integrity constraint becomes vacuous and only the second one applies, now requiring that `at(X,river_bank)` is false at time point `1` (for all instantiations of `X`). This mechanism ensures that final rules always apply exclusively to the last point in time. Note that the change of the truth value of external atoms via the API accounts for the addition of facts in (3).

4.4 Extracting traces from regular stable models

telingo translates a temporal logic program into a regular one, whose stable models are incrementally computed by *clingo*. The obtained model is then translated back into a temporal trace by reversing translation τ on the atoms in the model. That is, each atom a_k is turned into a and associated with the state numbered k.

5 Experiments

To check whether our approach imposes a significant burden on grounding and solving, we set up the following experiment: We took the benchmark suite used in [8] for incrementally solving ASP planning benchmarks.[12] This benchmark suite was obtained in [8] by manually producing incremental ASP encodings from encodings using a fixed plan length. This includes the benchmark domains *hanoi-tower*, *labyrinth*, *no-mystery*, *ricochet-robots*, *sokoban*, and *visit-all*, all originating in recent ASP competitions. In turn, we manually translated the incremental encodings from [8] into the temporal input language of *telingo*.[13] This resulted in two benchmark suites, in each case consisting of 69 benchmark instances. We then contrasted the results obtained by solving the incremental instances with *clingo* 5.3 and the temporal ones with *telingo* 1.0 (also based on *clingo* 5.3). The experiments ran under Linux on Intel Xeon E5-2650 v4 processors and 64 GB memory; we selected instances solvable within 24h by both *clingo* and *telingo*, and computed a single model.

Figure 1 shows two scatter plots comparing the runtime of *clingo* and *telingo* for both grounding and solving. Each point in such a plot displays the runtimes for one instance; the runtime of *clingo* is displayed on the x-axis and the runtime of *telingo* on the y-axis. Thus, we find easier instances in the lower left part and harder instances in the upper right part of a plot. Furthermore, the farther a point is away from the diagonal, the more the runtimes of both systems diverge; for points in the lower right part of a plot, *telingo* is faster and for points in the upper left part, *clingo* is faster. To highlight runtime differences, we use the colors from a heat map ranging from blue (both systems are equally fast) over yellow to red (highest runtime deviation of both systems).

[12] `https://github.com/potassco/asp-planning-benchmarks`

[13] `https://github.com/potassco/clingo-vs-telingo-planning/tree/v1.0.0`

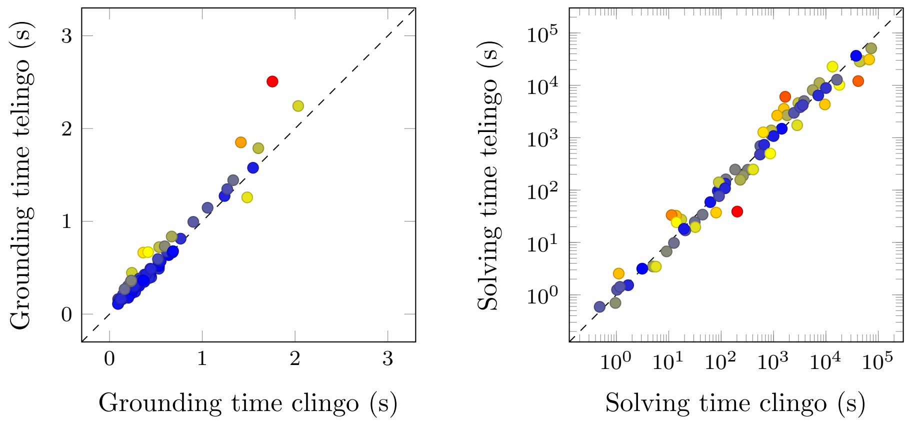

Fig. 1. Grounding and solving times of clingo and telingo

First, let us look at the grounding times in the left plot in Figure 1. We see that most points are close to the diagonal, showing that both systems perform quite similarly. All instances could be grounded in less than 3 seconds making grounding times negligible in the overall runtime. Furthermore, the resulting ground programs have nearly the same number of rules and atoms (each deviates by 0.01% on average). These results are not that surprising given that ASP planning benchmarks only deal with one-step transitions, and do not involve any complex temporal statements. Here, the translation in *telingo* boils down to adding time parameters to atoms. Hence, the translated program passed to the grounder is very similar to the incremental program used by *clingo*.

Next, we look at the solving times depicted in the right plot in Figure 1. Note that both *clingo* and *telingo* find solutions for the smallest horizon where a problem is satisfiable. Since the difficulty of planning problems increases exponentially with the search horizon, we use logarithmic axes in the plot. We see that there are runtime fluctuations between both systems. This is due to different traversals of the search space of both systems induced by heuristic effects. Such fluctuations are to be expected with ASP solvers, which are sensitive to small changes in instances, where even changing the order in which rules are passed to a solver can make a big difference in runtime.

All in all, our experiments confirm that *telingo*'s machinery imposes no significant encumbrance compared to a direct treatment with *clingo*.[14]

6 Discussion

We have described the temporal ASP system *telingo*, starting from its input language, over its workflow on top of *clingo*, to an empirical demonstration of its lightweight machinery. Previous temporal extensions to ASP [3, 12] relied on different semantics resulting in translations to automata and thus model checkers. What makes *telingo*

[14] Detailed results are obtainable at `https://github.com/potassco/clingo-vs-telingo-planning/tree/v1.0.0/benchmark-results`

interesting is that it constitutes a true extension of the ASP system *clingo*, and provides us with a full-fledged temporal modeling language. Moreover, it allows for an easy embedding of action languages (cf. [4]) and offers the specification of nested logic programs. For the future, we envisage the integration of constructs from dynamic logic, as proposed in [2], and the integration of more flexible reasoning modes, as used in [8].

Acknowledgments. This work was partially supported by MINECO, Spain, grant TIC2017-84453-P, Xunta de Galicia, Spain (GPC ED431B 2016/035 and 2016-2019 ED431G/01, CITIC), and DFG grant SCHA 550/9.

References

1. 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)
2. Bosser, A., Cabalar, P., Diéguez, M., Schaub, T.: Introducing temporal stable models for linear dynamic logic. In: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning. pp. 12–21. AAAI Press (2018)
3. Cabalar, P., Diéguez, M.: STELP: a tool for temporal answer set programming. In: [7], pp. 370–375
4. 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)
5. Cabalar, P., Pérez Vega, G.: Temporal equilibrium logic: A first approach. In: Proc. of the International Conference on Computer Aided Systems Theory. pp. 241–248. Springer (2007)
6. De Giacomo, G., Vardi, M.: Linear temporal logic and linear dynamic logic on finite traces. In: Proceedings of the International Joint Conference on Artificial Intelligence. pp. 854–860. IJCAI/AAAI Press (2013)
7. Delgrande, J., Faber, W. (eds.): Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning, Springer (2011)
8. Dimopoulos, Y., Gebser, M., Lühne, P., Romero, J., Schaub, T.: plasp 3: Towards effective ASP planning. Theory and Practice of Logic Programming (2018), to appear.
9. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Wanko, P.: Theory solving made easy with clingo 5. In: Technical Communications of the International Conference on Logic Programming. vol. 52, pp. 2:1–2:15. OASIcs (2016)
10. 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)
11. Gebser, M., Kaminski, R., König, A., Schaub, T.: Advances in gringo series 3. In: [7], pp. 345–351
12. Giordano, L., Martelli, A., Theseider Dupré, D.: Reasoning about actions with temporal answer sets. Theory and Practice of Logic Programming **13**(2), 201–225 (2013)
13. Kaminski, R., Schaub, T., Wanko, P.: A tutorial on hybrid answer set solving with clingo. In: Proc. of International Summer School of the Reasoning Web. pp. 167–203. Springer (2017)
14. Kamp, J.: Tense Logic and the Theory of Linear Order. Ph.D. thesis, UCLA (1968)
15. Lifschitz, V.: Answer set planning. In: Proceedings of the International Conference on Logic Programming. pp. 23–37. MIT Press (1999)
16. Oikarinen, E., Janhunen, T.: Modular equivalence for normal logic programs. In: Proceedings of the European Conference on Artificial Intelligence. pp. 412–416. IOS Press (2006)
17. Pnueli, A.: The temporal logic of programs. In: Proceedings of the Symposium on Foundations of Computer Science. pp. 46–57. IEEE Press (1977)