TPLP **26** (1-2): 180–196, 2026. © The Author(s), 2026. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution and reproduction, provided the original article is properly cited.

doi:10.1017/S1471068426100568 First published online 30 July 2026

*Meta-Programming for Linear-Time Temporal Answer Set Programming**

SUSANA HAHN, AMADÉ NEMES, JAVIER ROMERO and TORSTEN SCHAUB
Institute of Computer Science, University of Potsdam, Germany
(*e-mails:* hahnmartinlu@uni-potsdam.de, nemes@uni-potsdam.de, javier@cs.uni-potsdam.de, torsten@cs.uni-potsdam.de)

submitted 30 May 2026 UTC; revised 30 May 2026 UTC; accepted 1 June 2026 UTC

Abstract

The development of temporal extensions of Answer Set Programming (ASP) has led to the emergence of non-monotonic linear-time (TEL), dynamic (DEL), and metric (MEL) temporal equilibrium logics. However, the inherent rigidity of highly optimized ASP systems often hinders the rapid exploration and implementation of alternative logical designs. In this work, we propose a flexible meta-programming framework that operationalizes the semantics of varied temporal logics through a unified, declarative framework. Our approach extends standard ASP meta-programming by augmenting *clingo*'s theory grammar with formal type specifications and nesting capabilities. To ensure semantic correctness, we introduce a transformation pipeline that protects nested modalities from stable-model-based simplifications during grounding. We demonstrate the extensibility of our framework by implementing meta-encodings for TEL, MEL, and DEL. We provide a comprehensive account of TEL and highlight the key features for managing the interval constraints of MEL and the Fischer–Ladner closure in DEL. Finally, we introduce the *metasp* system, a versatile tool that encapsulates this workflow.

KEYWORDS: Answer Set Programming, meta-programming, linear-time temporal logic

1 Introduction

The logical foundations of Answer Set Programming (ASP; Lifschitz 2019) rest upon the Logic of Here-and-There and its non-monotonic extension, Equilibrium Logic (EL; Pearce 2006). Hence, extending ASP often necessitates amalgamating these foundations with complementary formalisms. In the temporal setting, this synergy has led to the emergence of linear-time (TEL; Aguado *et al*. 2023), dynamic (DEL; Cabalar *et al*. 2020), and metric (MEL; Becker *et al*. 2024b) temporal equilibrium logics. While the logic programming fragments of TEL and DEL have been realized in the monolithic *telingo* system (Cabalar *et al*. 2019), meta-encodings for metric variants have only recently emerged (Becker *et al*. 2024a). This shift, moving from hard-coded systems like *telingo* toward transparent meta-encodings, is motivated by the architectural rigidity inherent in highly optimized

* Research partially funded by DFG grant SCHA 550/15, Germany.

https://doi.org/10.1017/S1471068426100568 Published online by Cambridge University Press

ASP systems. While such systems excel in performance, their internal complexity often hinders the rapid implementation of logical extensions. In contrast, meta-programming provides an accessible and flexible environment that overcomes this limitation, enabling the declarative exploration of alternative designs through executable blueprints.

To realize this flexibility, meta-programming leverages reification: the process of representing a logic program's syntax as a set of facts. These facts are combined with a meta-encoding that implements the desired semantics by reinterpreting the original program's components. A major milestone in this area was the introduction of automated reification and a standardized format in *clingo* (Gebser *et al*. 2011), initially directed at augmenting optimization but later applied to broader semantic reinterpretations (Kaminski *et al*. 2023). Notably, these earlier approaches dealt with the standard ASP language, ensuring high compatibility with the existing toolchain. However, handling foreign language constructs via theory expressions currently introduces a significant "semantic gap." While *clingo* natively supports their reification, it produces fine-grained facts representing serialized syntax trees. Lifting these low-level structures into a format suitable for high-level meta-programming requires substantial manual effort. Furthermore, this approach lacks native support for nested theory expressions, restricts their syntactic occurrence, and has limited integration of syntactic checks on the well-formedness of such expressions.

To address these limitations, we significantly extend *clingo*'s meta-programming framework to support advanced constructs such as temporal modalities, path expressions, and intervals. By augmenting *clingo*'s theory grammar with recursive type specifications, we enable the unrestricted occurrence of these constructs, including within conditional literals and aggregates. A fundamental technical challenge in this endeavor is preventing the grounder from applying standard stable-model simplifications (cf. Section 3). Because simplifying nested modalities prematurely compromises semantic correctness, we introduce a transformation pipeline (Section 4) that insulates these expressions, preserving the structural integrity of the temporal program. We detail this approach in the context of TEL and subsequently address the challenges emerging in MEL and DEL in Sections 5 and 6. The resulting *metasp* system (Hahn *et al*. 2026; Section 7) is deeply integrated into a modern ASP ecosystem.[1] By leveraging tools such as *aspen-tree*, *clinguin*, *clorm*, and a custom *tree-sitter* grammar, it provides a robust meta-programming environment for both theoretical exploration and practical application. Finally, we evaluate its performance against the monolithic *telingo* system in Section 8 and conclude in Section 9.

2 Temporal logics at a glance

Linear-time temporal logics define temporal models as sequences of states. Following common conventions in AI (De Giacomo and Vardi 2013), we restrict our focus to finite sequences of states, which serve as the core semantic foundation for the temporal ASP extensions considered in this work. Standard linear-time logics (Pnueli 1977) incorporate temporal modal operators, such as next ($\circ$), always ($\Box$), and eventually ($\Diamond$). Specifically, $\circ\varphi$ requires that φ holds in the immediate successor state, while $\Box\varphi$ and $\Diamond\varphi$ demand that φ holds in all or some future states (including the current one), respectively. Furthermore, we utilize the initial-state operator $\mathbf{I}$, which holds exclusively in the first state of a

[1] https://potassco.org/aspen,clinguin,clorm,tree-sitter.

https://doi.org/10.1017/S1471068426100568 Published online by Cambridge University Press

```
1  time(0..n).

3  conjunction(B,T) :- literal_tuple(B), time(T),
4          hold(L,T) : literal_tuple(B, L), L > 0;
5      not hold(L,T) : literal_tuple(B,-L), L > 0.

7  body(normal(B),T) :- rule(_,normal(B)), conjunction(B,T), time(T).
8  body(sum(B,G),T)  :- rule(_,sum(B,G)), time(T),
9      #sum { W,L :     hold(L,T), weighted_literal_tuple(B, L,W), L > 0 ;
10            W,L : not hold(L,T), weighted_literal_tuple(B,-L,W), L > 0 } >= G.

12   hold(A,T) : atom_tuple(H,A)   :- rule(disjunction(H),B), body(B,T), time(T).
13 { hold(A,T) : atom_tuple(H,A) } :- rule(     choice(H),B), body(B,T), time(T).
```

Listing 1. Timed meta-encoding (`teta.lp`)

sequence. Additional operators can be derived from these primitives; for example, the final-state operator $\mathbf{F} \equiv \neg\circ\top$ only holds at the terminal state of a finite sequence. More formally, a temporal model $(X_i)_{i=0}^n$, consisting of a sequence of $n{+}1$ interpretations, satisfies a formula formed with our example operators at time step $t \in \{0, \ldots, n\}$, if

$$(X_i)_{i=0}^n, t \models \mathbf{I} \quad \text{iff } t = 0, \tag{1}$$

$$(X_i)_{i=0}^n, t \models \circ\varphi \text{ iff } t < n \text{ and } (X_i)_{i=0}^n, t{+}1 \models \varphi, \tag{2}$$

$$(X_i)_{i=0}^n, t \models \Diamond\varphi \text{ iff } (X_i)_{i=0}^n, j \models \varphi \text{ for some } j \in \{t, \ldots, n\}. \tag{3}$$

In dynamic logic (Pratt 1976), the modal operators $\Box$ and $\Diamond$ are parametrized by regular expressions that define paths over sequences of states. Specifically, the dynamic formulas $[\rho]\,\varphi$ and $\langle\rho\rangle\,\varphi$ require that φ holds in all or some of the states, respectively, reachable via a path segment satisfying the expression ρ. Similarly, in metric temporal logic (Alur and Henzinger 1992), $\Box$ and $\Diamond$ are parametrized by a time interval, which, in our context, is defined over natural numbers. The metric formulas $\Box_{[l..u)}\varphi$ and $\Diamond_{[l..u)}\varphi$ stipulate that φ holds in all or some states, respectively, whose associated time point lies within the interval $[t{+}l..t{+}u)$, where t denotes the current time point.

While these semantic intuitions have been extensively explored across various logical contexts, our work centers on adapting them to the non-monotonic realm of Equilibrium Logics. In particular, we target fragments of TEL, MEL, and DEL that are amenable to implementation via ASP. To bridge this theoretical foundation with a practical implementation, we now establish the semantic infrastructure required to represent discrete state sequences within *clingo*'s meta-programming framework.

3 Setting the stage for temporal meta-programming

To capture the series of states in linear temporal models, we present in Listing 1 a meta-encoding that extends the one for plain ASP outlined in (Kaminski *et al.* 2023). This extension relies on a time index provided by the atom `time(T)` and the inclusion of the variable `T` as the final argument in the encoding's primary predicates. The set of instantiated rules sharing a common time index effectively duplicates the behavior of the plain ASP meta-encoding. The number of duplicated ASP encodings is given by the parameter `n`; see Line 1.

https://doi.org/10.1017/S1471068426100568 Published online by Cambridge University Press

Prior to detailing the encoding in Listing 1, we must establish the underlying reified fact format. For instance, *clingo*'s reification transforms the (non-temporal) rule

```
red(l1) :- not green(l1), light(l1).
```

into the following set of facts (omitting definitions for `green(l1)` and `light(l1)`):

```
1 rule(disjunction(0),normal(0)).
2 atom_tuple(0).          atom_tuple(0,3).
3 literal_tuple(0).       literal_tuple(0,-2).   literal_tuple(0,1).
4 output(light(l1),1).    literal_tuple(1).      literal_tuple(1,1).
5 output(green(l1),2).    literal_tuple(2).      literal_tuple(2,2).
6 output(red(l1),3).      literal_tuple(3).      literal_tuple(3,3).
```

The syntactic structure of the rule is captured by the reified facts in Lines 1–3. The primary structure is defined by the instance of `rule/2` in Line 1, which specifies a disjunctive head and a normal body, both identified by the constant `0`. In this format, *clingo* represents heads and bodies as tuples of atoms and literals, respectively, utilizing the predicates `atom_tuple` and `literal_tuple`. Within these tuples, ground literals are mapped to unique integer identifiers. For instance, the rule's head consists of a single atom represented in Line 2 as the tuple `(3)`, while the body is represented in Line 3 as a conjunction of two literals forming the tuple `(-2,1)`. Finally, the `output/2` facts in Lines 4–6 establish the necessary mapping between human-readable symbolic names and these internal integer identifiers. For example, `output(light(l1),1)` links the symbolic atom to the integer `1` via the corresponding literal tuple `(1)`.[2]

Notably, *clingo*'s reification process is subject to stable-models-preserving simplifications that occur during grounding. Consequently, the resulting set of reified facts reflects the simplified rule structure rather than the original program. If the truth value of a body literal is established during the grounding phase, that literal is removed from the rule and, by extension, from its reified representation. For instance, this occurs if a negative literal ‘‘`not green(l1)`’’ is satisfied because `green(l1)` is non-derivable. Conversely, if a rule's body is determined to be unsatisfiable, the entire rule becomes vacuous and is eliminated, along with its associated reified facts. This scenario arises, for example, if ‘‘`not green(l1)`’’ is found to be false because `green(l1)` has been established as true.

Following the reification of the original program, the resulting facts are combined with a meta-encoding and processed by an ASP system. In the meta-encoding presented in Listing 1, the truth of a literal at a specific time step is represented by instances of the `hold/2` predicate. Accordingly, the rule in Lines 3–5 derives an instance of `conjunction(B,T)` for a time step `T`, if all literals within the associated `literal_tuple(B)` hold at that time. This enables the derivation of `body(normal(B),T)` in Line 7 whenever the literal tuple `B` represents a normal rule body. Consequently, the head atoms of the corresponding rule are derived in Lines 13 or 13. For instance, using our previous example facts and assuming that we have `time(0)` and `hold(1,0)` but not `hold(2,0)`, we derive `conjunction(0,0)` and `body(normal(0),0)`, ultimately concluding `hold(3,0)` via the rule in Line 12. For a comprehensive account of *clingo*'s reification, we refer to (Kaminski *et al*. 2023).

[2] This indirection is caused by *clingo*'s normalized show statements that use conditional terms.

https://doi.org/10.1017/S1471068426100568 Published online by Cambridge University Press

```
1  #type tel {
2      subtypes:    { atom; };
3      expressions: {
4          &true/0; &not/1: { type: tel, safety: unsafe; };
5          &initial/0;
6          &next/1:         { type: tel, safety:   safe; };
7          &eventually/1:   { type: tel, safety:   safe; }; };
8      macros:      { &final: &not(&next(&true)); };
9      occurrence: any;
10 }.
```

Listing 2. Abridged linear-time temporal grammar (`tel-grammar.lp`).

Thus far, our meta-encoding captures the program's behavior at each time step independently, effectively yielding a sequence of isolated state valuations. In the following, we describe how temporal operators are utilized to establish the necessary logical connections between these states, thereby defining the resulting temporal stable models.

4 Temporal meta-programming

To illustrate our approach, we consider a traffic light system modeled with linear-time temporal operators in (4) to (6). In this scenario, a traffic light is red by default, unless it is green. When a pedestrian button is pushed, the light becomes eventually green from the next step on. The button for light l_1 is pushed one step after the initial state.

$$\Box(\neg green(L) \rightarrow red(L)) \tag{4}$$

$$\Box(push(L) \rightarrow \circ\Diamond green(L)) \tag{5}$$

$$\circ push(l_1) \tag{6}$$

In TEL, these constraints yield no temporal models of length one and two, but one of length three, given by the state sequence: $\langle\{red(l_1)\}, \{red(l_1), push(l_1)\}, \{green(l_1)\}\rangle$.

4.1 Syntax definition

To define the additional syntax required for temporal expressions, we extend the theory language of *clingo* by `#type` definitions.[3] A representative grammar for the type `tel` is given in Listing 2, which serves to augment standard *clingo* atoms with temporal formulas whose arguments remain regular *clingo* terms. This extension is facilitated by the `subtypes` field in Line 2, permitting the inclusion of other types, such as the predefined type `atom` for standard *clingo* atoms, into our temporal expressions. Consequently, a standard atom like `green(l1)` is treated as a `tel` expression and functions as a valid argument of type `tel` within nested temporal formulas. Other supported base types include `number`, `string`, `infimum`, and `supremum`.

Lines 3–8 define the formation of temporal `expressions` incorporating the $\top$, $\neg$, **I**, $\circ$, $\Diamond$, and **F** operators, each prefixed by `amp` in accordance with *clingo*'s theory grammar

[3] *Clingo* directives are preceded with `#`.

https://doi.org/10.1017/S1471068426100568 Published online by Cambridge University Press

```
1  light(l1).
2  red(L) :- not green(L), light(L).
3  &next(&eventually(green(L))) :- push(L).
4  &next(push(l1)) :- &initial.
```

Listing 3. Linear-time temporal example (`telex.lp`).

conventions. To support nesting, the specification is defined recursively, allowing constructs such as `& true` or `& not` (Line 3) to be embedded within other expressions. This recursive design facilitates the customization of existing constructs and enables the seamless addition of new operators, such as a Boolean `& or`. Unlike standard theory atoms, these temporal formulas can be nested and utilized within conditional literals or aggregates. For example, the conditional `‘‘ & eventually(green(L)) : light(L)’’` concisely stipulates that "every light must eventually turn green." Expressions are further refined by specifying the `type` and the `safety` condition for their arguments, which are essential for subsequent transformations. For instance, the arguments for `& next` and `& eventually` are of type `tel` and are declared as `safe`, whereas the argument for `& not` is declared as `unsafe`. Indeed, the latter declaration can be omitted, as unspecified `safety` conditions default to `unsafe`.

Defined operators can be specified with `macros`. In Line 8, for example, we define the final-state operator **F** as $\neg\circ\top$, which motivates the inclusion of `& true` and `& not` in the base specification. For more complex operators with arguments, placeholders can be used, as detailed in Section 5 and 6. Furthermore, the `occurrence` keyword identifies the valid syntactic positions for expressions of a given type, namely `any`, `head`, `body`, or `directive`. While `tel` expressions are permitted in both heads and bodies of rules (`any`), omitting this declaration restricts a type to serving exclusively as an argument for other expressions. This distinction is essential for defining auxiliary types, like intervals in Section 5, which act as structural components rather than standalone atoms.

Using the grammar in Listing 2, we can express the rules in (4) to (6) as in Listing 3. For simplicity, each rule is implicitly governed by the "always" operator $\Box$, meaning it applies at every time step of the temporal model. Rules restricted to the initial state, like (6), are qualified by the `& initial` operator (representing **I**) within their body to anchor them to the first state of the model.

4.2 Non-ground transformations

Logic programs containing type expressions undergo several transformations to leverage standard grounding technology without compromising semantic correctness. The initial phase converts programs with temporal type expressions into standard programs parsable by *clingo*. This transformation is designed to be partially reversible during output processing to maintain readability.

The primary objective of our non-ground transformations is to avoid incorrect simplifications during grounding. To achieve this, the system instruments the input program with `#external` statements. As illustrated in Listing 4, a directive such as "`#external push(l1).’’` in Line 8 instructs the grounder to exempt that specific atom from simplifications. Without this protection, the grounder would erroneously assume that expressions

https://doi.org/10.1017/S1471068426100568 Published online by Cambridge University Press

```
6  #external green(L) : push(L).
7  #external &initial.
8  #external push(11).
```

Listing 4. Result of (non-ground) transformation of Listing 3.

like `& initial` or instances of `push/1` and `green/1` are non-derivable and therefore false. This would lead to semantically incorrect models, such as a traffic light remaining red indefinitely.

The generation of these protective directives relies on an extended definition of safety. An atom occurrence in a rule body is deemed *safe*, if it is neither within the scope of negation nor within the scope of an argument of a theory expression declared as `unsafe` in the grammar. A rule is considered safe if all its variables occur in safe body atoms. For example, the rule ‘‘`wait(L) :- &eventually(green(L)), not green(L).`’’ is safe because the variable `L` is bound by the occurrence of `green(L)` within the first conjunct. This occurrence is safe because the single argument of `& eventually/1` is declared as `safe` in the grammar and is not negated. Conversely, if a variable only appears within negated literals, such as ‘‘`not green(L),`’’ or within theory expressions declared as `unsafe` (e.g., `& not(green(L))`), the rule is deemed unsafe.

In our system, safety verification is interleaved with the generation of external directives. For each safe rule, *metasp* generates two distinct types of `#external`. The first type prevents the grounder from discarding rules with body expressions that might be deemed non-derivable. The system adds one `#external` statement for each temporal expression in the body, conditioned by the rule's safe body atoms to ensure correct variable binding. For instance, our `wait` example yields ‘‘`#external &eventually(green(L)): green(L).`’’ Similarly, the statement in Line 7 is induced by the rule in Line 4. The second type ensures that atoms within head expressions are explicitly incorporated into the grounder's instantiation domain. The system generates an `#external` directive for every atom appearing within a non-negated head expression, again conditioned on safe body atoms to maintain correct variable bindings. As illustrated in Line 6 of Listing 4, this ensures that atoms like `green(L)` remain part of the grounding domain even when nested within modalities like `& eventually`. This incorporation of `#external` directives allows for a complete ground instantiation in a single call to *clingo*, representing a significant advancement over previous two-pass temporal grounding methods (Cabalar *et al*. 2010); its formal correctness is established in (Nemes 2024).

Declaring `safety` conditions for expression arguments is critical to theory grammar design. A sufficient criterion is whether the satisfaction of a formula implies the satisfaction of its argument at some time step. In TEL, the arguments of `& next/1` and `& eventually/1` meet this criterion and are thus declared `safe`. When this fails, the argument is better declared `unsafe`. For instance, Nemes (2024) shows that the single arguments of $\circ$, $\Diamond$, and $\Box$ provide safe bindings in TEL (as do the second arguments of “until” and “release”).[4]

Finally, `#show` directives are transformed into internal rules and subsequently reified. This process provides a comprehensive symbol table, expressed through `output/2` facts, and yields specialized facts such as `show_atom/2` and `show_term/2` that capture the

[4] Interestingly, this is different in DEL, where path expressions may interfere (cf. Listing 11 of Section 6).

https://doi.org/10.1017/S1471068426100568 Published online by Cambridge University Press

```
1  output(light(l1),6).     output(&next(push(l1)),7).
2  output(green(l1),2).     output(&next(&eventually(green(l1))),8).
3  output(red(l1),4).       output(&initial,5).
4  output(push(l1),1).
6  formula(atom,light(l1)).              formula(tel,light(l1)).
7  formula(atom,red(l1)).                formula(tel,red(l1)).
8  formula(atom,push(l1)).               formula(tel,push(l1)).
9  formula(atom,green(l1)).              formula(tel,green(l1)).
10 formula(tel,&eventually(green(l1))).
11 formula(tel,&next(&eventually(green(l1)))).
12 formula(tel,&next(push(l1))).
13 formula(tel,&initial).
```

Listing 5. Reified temporal example.

syntactic conditions of the original directives. By leveraging these reified statements, users can define declarative output policies to customize the output of complex temporal formulas. We refer the reader to (Hahn *et al.* 2026) for a detailed account of this mechanism.

4.3 Extended reification

Once non-ground transformations are complete, the program is grounded and reified using an extension of *clingo*'s reification process designed for temporal formulas. During this phase, type checking and operator expansion are interleaved with grounding, allowing the system to record each subformula and its associated type through corresponding `formula/2` facts. In our example, the resulting subformulas are detailed in Lines 6–13 of Listing 5; for instance, Lines 9–11 represent the reified subexpressions derived from `& next( & eventually(green(l1)))`. Notably, a single subformula may appear in multiple `formula/2` facts, depending on its type memberships. For example, `green(l1)` is simultaneously categorized as an `atom` and a temporal expression (`tel`). The `formula/2` predicate serves as a domain predicate in our meta-encodings, effectively constraining the instantiation of formula-based rules. This representation allows us to easily incorporate advanced syntactic concepts, such as the closure of Fischer and Ladner (1979) in Section 6, by simply adding auxiliary rules to define extended subexpression relationships.

Standard symbol table information is retained via the `output/2` predicate. Because our non-ground transformations instrument the program by inserting `#external` directives and removing `#show` statements, the grounder is prevented from aggressively simplifying the program, and all atoms and temporal formulas are preserved as `output/2` facts. The symbol table obtained in our example is given in Lines 1–4 of Listing 5.

The core of this reified representation utilizes standard reification predicates: a `rule/2` instance for each rule in the original program is provided alongside the associated `atom_tuple` and `literal_tuple` facts that capture their respective heads and bodies as outlined in Section 3. In addition to the `output/2` and `formula/2` facts, the representation incorporates `external/2` facts, as well as `show_term/2` and `show_atom/2` instances. These correspond to the injected `#external` directives and transformed `#show` statements. For the sake of brevity, these specific predicate instances are omitted from the discussion.

https://doi.org/10.1017/S1471068426100568 Published online by Cambridge University Press

```
1  true(O,T) :- output(O,B), time(T), hold(L,T): literal_tuple(B,L).
2  hold(L,T) :- output(O,B), time(T), true(O,T), literal_tuple(B,L).

4  true(&initial,T) :- formula(tel,&initial), T = 0.
5            T = 0 :- formula(tel,&initial), true(&initial,T).

7  true(&next(F),T) :- formula(tel,&next(F)), T < n, true(F,T+1).
8            T < n :- formula(tel,&next(F)), true(&next(F),T).
9       true(F,T+1):- formula(tel,&next(F)), true(&next(F),T).

11   true(F,J) : J=T..n :- formula(tel,&eventually(F)), true(&eventually(F),T).
12 true(&eventually(F),T) :- formula(tel,&eventually(F)),
13                           J=T..n, true(F,J), time(T).
```

Listing 6. Encoding of TEL semantics of ❙, ○, and ◊ (`tel-semantics.lp`).

4.4 Temporal meta-encoding

A meta-encoding operationalizes the semantics of a logic program by defining the underlying semantic structure and interpreting the program's components within that structure. The design of our meta-encoding reflects this bipartite nature: the first part (Listing 1) establishes a sequence of independent stable models, where temporal formulas are treated as standard atoms, while the second (Listing 6) accounts for the satisfaction relation of temporal operators ❙, ○, and ◊.

The first part of our meta-encoding employs the predicate `hold/2` to represent the valuation of literals along different time steps; this representation captures linear-time temporal models and is thus common to the temporal logics TEL, MEL, and DEL.

The transition between both parts relies on the `true/2` predicate. While `hold/2` refers to atoms and formulas via numeric identifiers, `true/2` utilizes their symbolic representations. This distinction is vital, as it enables the evaluation of operators based on their inherent syntactic structure rather than opaque identifiers. To this end, the `hold/2` atoms are related to nested temporal formulas through the `true/2` predicate in Listing 6 to capture the inductive definition of satisfaction. The rules in Lines 1–2 establish the base case by synchronizing `hold/2` and `true/2` instances for all elements in the symbol table. Whenever a symbolic representation `o` corresponds to a numeric identifier `l`, the system typically relies on the presence of the facts `output(o,b)` and `literal_tuple(b,l)` to enforce the equivalence between `true(o,t)` and `hold(l,t)` along all time steps `t`. However, a subtle distinction arises when a formula `o` appears as a fact: because it lacks an associated rule body, the corresponding `literal_tuple/2` is absent from the reified output. This is elegantly handled by the rule in Line 1 via a conditional, which allows for deriving `true(o,t)` directly from the symbol table, while the first part of the meta-encoding independently establishes the truth of the corresponding fact `hold(l,t)`.

Building on this, the satisfaction relations for ❙, ○, and ◊ are implemented by the subsequent rules in Listing 6. The individual cases, and thus the overall grounding process, is governed by `formula/2`, acting as a domain predicate. Each group of rules in Lines 4–5, 7–9, and 11–13 establishes a logical equivalence that captures the formal truth conditions for ❙, ○, and ◊ as defined in (1), (2), and (3), respectively. For instance, the satisfaction of $\Diamond\varphi$ in (3) can be expressed in our framework as `true( & eventually(F),T)` iff `true(F,j)` for some $\mathtt{j} \in \{\mathtt{T}, \ldots, \mathtt{n}\}$. This bidirectional relationship is realized in Lines 11 and 13, whose rules collectively enforce the semantics of the "eventually" operator.

https://doi.org/10.1017/S1471068426100568 Published online by Cambridge University Press

```
1  #type ub { subtypes: { number; supremum; }; }.
2  #type interval { expressions: { &i/2: { type: number; type: ub; }; }; }.
3  #type mel {
4   subtypes:     { atom; };
5   expressions:  {
6    &initial/0;
7    &next/2:       {type: interval, safety: unsafe; type: mel, safety: safe;};
8    &eventually/2: {type: interval, safety: unsafe; type: mel, safety: safe;};
9   };
10  macros:       { &next(v): &next(&i(0,#sup), v); where: {v: mel;}; };
11  occurrence: any;
12 }.
```

Listing 7. Abridged metric temporal grammar.

Taken together, the theory grammar in Listing 2 and the meta-encodings in Listing 1 and 6 provide a complete framework for evaluating our temporal program presented in Listing 3. More generally, given a temporal logic program P, and $n \geq 0$, we get a 1–1 correspondence between the temporal models of length n of P and the stable models of the reification of P, obtained using Listing 2, together with Listings 1 and 6. The broader functionality of our tool, *metasp*, is discussed in Section 7.

5 Metric meta-programming

Having established our meta-programming workflow through TEL, we now demonstrate the framework's extensibility within the more complex setting of MEL. This section illustrates how our approach handles quantitative temporal constraints by leveraging a modular grammar and hybrid solving techniques.

The grammar for this metric extension, detailed in Listing 7, incorporates the auxiliary types `interval` and `ub` to facilitate the representation of temporal intervals. Intervals are constructed using the binary operator `& i`, which pairs a numeric lower bound with an upper bound of type `ub` (Line 2). As specified in Line 1, this upper bound may be either a specific `number` or the constant `#sup`, the sole member of the predefined `supremum` type. Both types are predefined in *metasp* and serve purely as structural components. In fact, while expressions of type `mel` are permitted to occur in rule heads and bodies (as declared in Line 11), the `interval` and `ub` types are restricted from appearing as standalone atoms. This is enforced by the absence of an `occurrence` declaration for these types. Furthermore, both arguments of the interval operator `& i` are considered `unsafe`, as reflected by the lack of explicit `safety` declarations in the grammar.

Following the pattern of TEL expressions, Line 5 to 8 define formulas involving expressions with $\mathbf{I}$, $\circ_{[l..u)}\varphi$, and $\Diamond_{[l..u)}\varphi$. The latter two are captured by the binary operators `next/2` and `eventually/2`, which pair a metric `interval` with a `mel` formula. Similar to TEL, the second argument of type `mel` is considered `safe`, while the first, `interval` argument is declared `unsafe`. To streamline the syntax, Line 10 introduces a unary variant of the `& next` operator representing the unconstrained metric operator $\circ_{[0..\omega)}$. In this context, the interval imposes no constraints, and the operator amounts to its qualitative counterpart $\circ$ in TEL. This allows us to write `& next(F)` as a shorthand for the more verbose `& next( & i(0,#sup),F)`, where `F` is a `mel` formula. To facilitate this in

https://doi.org/10.1017/S1471068426100568 Published online by Cambridge University Press

```
1  light(l1).
2  red(L) :- not green(L), light(L).
3  &eventually(&i(10,15),green(L)):- push(L).
4  &next(push(l1)) :- &initial.
```

Listing 8. Metric temporal example.

```
1  &sum{ t(0) } = 0.
2  &sum{ t(T); -t(T-1) } >= 1 :- T = 1..n.
```

Listing 9. Encoding of the timing function τ.

```
1  true(&next(&i(L,U),F),T) :- formula(mel,&next(&i(L,U),F)), time(T),
2    true(F,T+1), &sum{t(T+1); -t(T)} >= L, &sum{t(T+1); -t(T)} < U, T < n.
```

Listing 10. Encoding for MEL semantics of $\circ_{[l..u)}$.

the theory specification, placeholders are utilized; these are specific constants that are declared and assigned types within a `where` clause.

The metric counterpart of our traffic lights example is provided in Listing 8. This encoding differs from Listing 3 primarily in Line 3, where the plain eventually head atom is replaced by its metric counterpart `& eventually( & i(10,15),green(L))`, stipulating that the light must turn green between time points 10 and 15. The occurrence of `& next(push(l1))` in Line 4 is replaced during reification by `& next( & i(0,#sup),push(l1))`.

To accommodate MEL's dual nature, combining qualitative temporal progression with quantitative timing constraints, our meta-encoding utilizes the hybrid ASP solver *clingcon* (Banbara *et al*. 2017) to leverage its native support for linear constraints over integers. Formally, each state X_t in a temporal model $(X_i)_{i=0}^n$ is associated in MEL (Becker *et al*. 2024b) with a discrete time point via a timing function τ satisfying $\tau(0) = 0$ and $\tau(t) - \tau(t-1) \geq 1$ for $t \in \{1, \ldots, n\}$. This relationship is operationalized in Listing 9 using *clingcon*'s `& sum` theory atom, in which each instance of functor `t/1` takes a non-negative integer value corresponding to a specific time point.

We illustrate the meta-encoding of metric operators in terms of the metric next $\circ_{[l..u)}$. As with TEL, the encoding is derived from semantic principles: A temporal model $(X_i)_{i=0}^n$ with a timing function τ satisfies $\circ_{[l..u)}\varphi$ at time step $t \in \{0, \ldots, n\}$, written $(X_i)_{i=0}^n, t \models \circ_{[l..u)}\varphi$, if

$$(X_i)_{i=0}^n, t{+}1 \models \varphi \text{ and } \tau(t+1) - \tau(t) \in [l..u) \text{ for } t < n \ .$$

We can express this within our framework as: `true(&next(&i(L,U),F),T)` iff `true(F,T+1)`, `&sum t(T+1); -t(T) >= L`, `&sum t(T+1); -t(T) <U`, and `T <n`. The right-to-left direction of this equivalence is given in Listing 10. The left-to-right direction is encoded analogously with four rules, one for each conjunct. The full encoding can be found at (Hahn *et al*. 2026).

6 Dynamic meta-programming

Lastly, we demonstrate the extensibility of our approach through DEL. The grammar for this extension, provided in Listing 11, inherits the TEL type definitions via the

https://doi.org/10.1017/S1471068426100568 Published online by Cambridge University Press

```
1  #type del{
2   subtypes:    { tel; };
3   expressions: {
4    &always/2:     {type: path, safety: unsafe; type: del, safety: unsafe;};
5    &eventually/2: {type: path, safety:   safe; type: del, safety:   safe;};};
6   occurrence:  any;
7  }.
8  #type path {
9   expressions: {
10   &step/0;
11   &test/1:   { type: del,  safety:   safe; };
12   &star/1:   { type: path, safety: unsafe; };
13   &choice/2: { type: path, safety: unsafe; type: path, safety: unsafe; };
14   &seq/2:    { type: path, safety:   safe; type: path, safety:   safe; }; };
15  macros:      {  a: &seq(&test(a), &step); where: { a: atom; }; };
16 }.
```

Listing 11. Abridged dynamic temporal grammar.

```
1  formula(del,&eventually(P,&eventually(&star(P),F))) :- formula(del,&eventually(&star(P),F)).
2  formula(del,&eventually(P1,F)) :- formula(del,&eventually(&choice(P1,P2),F)).
3  formula(del,&eventually(P2,F)) :- formula(del,&eventually(&choice(P1,P2),F)).
4  formula(del,&eventually(P1,&eventually(P2,F))) :- formula(del,&eventually(&seq(P1,P2),F)).
5  formula(del,&eventually(P2,F)) :- formula(del,&eventually(&seq(P1,P2),F)).
```

Listing 12. Additional rules for the Fischer–Ladner closure of `&eventually` formulas.

`subtypes` field.[5] Path expressions, defined in Lines 9–15, serve exclusively as arguments for the dynamic modal operators. These paths incorporate standard constructors, such as step (τ), test ($?$), star ($*$), choice ($+$), and sequence ($;$), to define regular expressions over state sequences. The macro introduced in Line 15 permits an atom a to be used directly in path expressions by translating it into the expression $(a?;\tau)$, a replacement strictly confined to the context of `path` expressions.[6]

For example, the formula $\mathbf{I} \wedge \langle (green(l_1); red(l_1))^* \rangle \mathbf{F}$ describes temporal models where light l_1 alternates between green and red. This is expressed in our framework as an integrity constraint:

```
:- &initial, not &eventually(&star(&seq(green(L),red(L))),&final).
```

This constraint effectively utilizes the `tel` operators `& initial` and `& final` alongside the new dynamic constructs. The instances of `green(L)` and `red(L)` are expanded during reification into `& seq( & test(green(L)), & step)` and `& seq( & test(red(L)), & step)`.

In DEL, the satisfaction relation has a scope that extends beyond the subformulas explicitly present in the input. One must also capture the recursive interplay between dynamic modal operators and path expressions, which requires a superset of the subformulas referred to as the Fischer and Ladner (1979) closure. In our framework, this closure is obtained declaratively by extending the definition of the `formula/2` predicate, as demonstrated for `& eventually` in Listing 12. For instance, Line 1 unfolds the "star" operator by one repetition, while Line 4 decomposes a sequence into nested modalities. This declarative expansion of the subformulas ensures that the meta-encoding can verify path satisfaction through a finite set of subexpressions derived entirely via ASP rules. The full encoding can be found at (Hahn *et al.* 2026).

[5] Alternatively, `del` and `tel` types can be defined next to each other (or even one redefined in the other).
[6] Note that DEL follows the single-sorted approach of LDL_f (De Giacomo and Vardi 2013).

https://doi.org/10.1017/S1471068426100568 Published online by Cambridge University Press

```
UNIX> metasp solve clingo telex.lp                    \
      --syntax-encoding tel-grammar.lp                \
      --semantics-encoding teta.lp tel-semantics.lp   \
      --printer temporal_printer -c n=2
Reading from telex.lp
Solving...
Answer: 1
 State 0: light(l1) red(l1)
 State 1: light(l1) push(l1) red(l1)
 State 2: green(l1) light(l1)

SATISFIABLE
```

Listing 13. System output showcasing the TEL setting.

7 The *metasp* system

The *metasp* system encapsulates our meta-programming workflow, providing a unified interface for defining and solving temporal ASP extensions. The system is implemented via an `Application` class that extends the target system's command line, ensuring full access to native options, such as optimization.

The `solve` command orchestrates the entire workflow from syntax parsing to model generation. As shown in Listing 13, it requires a target solver, typically *clingo*, or *clingcon* for metric extensions, alongside problem-specific and logic-specific encodings.

In Listing 13, we `solve` our temporal example in file `telex.lp` (Listing 3) with `clingo`. The options `--syntax-encoding` and `--semantics-encoding` are used to provide the grammar and meta-encodings for the desired logic. Users can customize the output via the `--printer` option. For example, the built-in `temporal_printer` organizes atoms by state to improve readability of temporal models. If omitted, the target system's default printer is used, though users may also provide custom Python scripts for specialized output policies. Standard *clingo* arguments, such as `''-c n=2''` for assigning constant values, are supported directly. The output in Listing 13 reflects a single temporal model, where the light becomes *green* at state 2, as expected.

To facilitate development, *metasp* allows users to intercept the transformation process at individual stages: the `transform` command prints the given program after initial first-order transformations, while `reify` outputs the extended reification facts. Unlike `solve`, these diagnostic commands require only grammar definitions and constants, bypassing the need for a back-end solver. For interactive solution exploration, the `ui` command launches an interface using `clinguin` (Beiser *et al*. 2024). To maintain a clean command-line experience, extension-specific options can be aggregated into configuration files. Finally, *metasp* supports multiple log levels and issues proactive warnings for common development issues, such as type mismatches or macro redefinitions.

8 Experiments

We conducted an empirical evaluation to assess the performance trade-offs between the flexible meta-programming framework of *metasp* and the specialized architecture of *telingo*. Our comparison focuses on TEL and DEL, as these are currently the only

https://doi.org/10.1017/S1471068426100568 Published online by Cambridge University Press

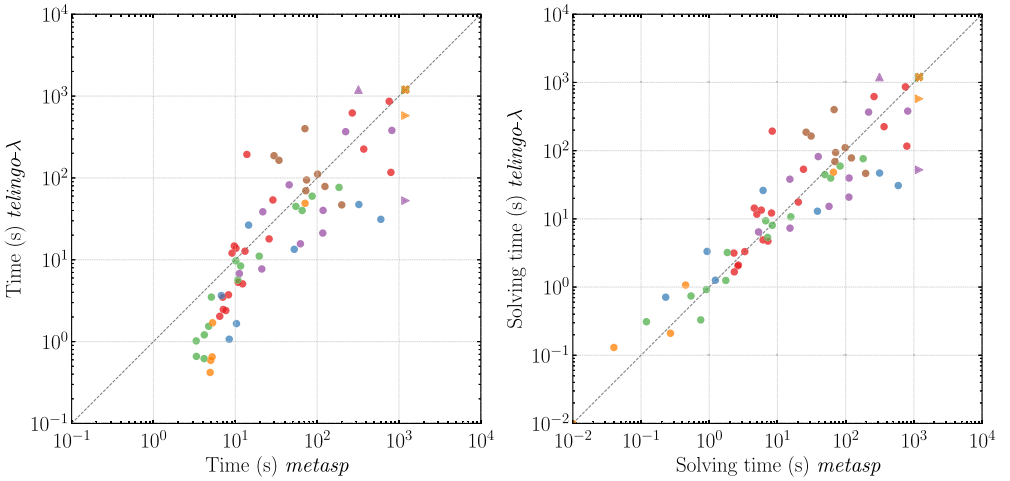


Figure 1. Comparison of runtime and solving time of *metasp* vs *telingo-λ* on TEL problems.

temporal extensions supported by *telingo*. All experiments were conducted on Debian Linux 10 machines with an Intel Xeon E5-2650v4 CPU (2.9 GHz) and 64 GB of RAM. We compare *metasp* 1.0, utilizing the meta-encodings and grammars partly introduced above with *clingo* 5.8, against a specialized fork of *telingo* 2.1.3. Individual runs were constrained by a time limit of 20 min and a memory limit of 24 GB.

Telingo solves TEL and DEL problems incrementally, gradually expanding the horizon. Because our meta-encodings evaluate a fixed horizon, the repeated unsatisfiability checks of the incremental approach would obscure a direct performance comparison. Therefore, we developed *telingo-λ*, a specialized fork of *telingo* that supports fixed-horizon solving.[7]

Our TEL evaluation utilizes 69 temporal planning instances from (Cabalar *et al*. 2019), spanning classic domains: *hanoi-towers* (red), *labyrinth* (blue), *no-mystery* (green), *richochet-robots* (purple), *sokoban* (orange), and *visitall* (brown). The results are visualized in Figure 1, where timeouts for *telingo-λ* and *metasp* are indicated by upward and rightward arrows, respectively; crosses indicate joint timeouts.

A clear performance trend emerges: on easier instances, *telingo-λ* exhibits faster total runtimes, but as complexity increases, the systems perform competitively. The solving time graph (right) clarifies this discrepancy. Here, the systems compete head-to-head, indicating the runtime difference is primarily driven by an initial grounding overhead in *metasp*. This penalty is expected, as *metasp* requires two distinct grounding phases: first to reify the input program, and second to instantiate the meta-encoding. Crucially, because time is not explicitly represented in TEL, the initial reification step does not scale with the horizon. Consequently, this overhead remains relatively flat and becomes negligible on more difficult instances requiring longer horizons.

Next, we evaluate *metasp* in the context of DEL using the elevator benchmark from (Levesque *et al*. 1997). While we adopt the general experimental setup of Cabalar *et al*. (2020), our evaluation employs a more recent version of *telingo*, which accounts for variations in the reported statistics. This experiment analyzes the impact of DEL-based

[7] Available at https://potassco.org/telingo.

https://doi.org/10.1017/S1471068426100568 Published online by Cambridge University Press

constraints, designed to eliminate redundant elevator movements, on the number of temporal traces, search choices, and final solver constraints. The full experiment comprises several elevator configurations, serving 5, 7, 9, and 11 floors, each evaluated over five different horizons. To illustrate the different behaviors of the systems, we discuss one representative instance in detail: the 11-floor configuration, which admits two optimal traces of length 17, evaluated at horizon 21. In the baseline configuration without control rules, the instance yields 200900 temporal traces. To compute these, *telingo*-λ and *metasp* explore search spaces of comparable size (21,6908 *vs.* 21,4840 choices) and produce a similar number of constraints (3221 *vs.* 3508). Upon introducing the DEL control rule, the number of traces drops to 2, aligning with the aforementioned optima. While both systems see a massive search space reduction, *telingo*-λ prunes more aggressively (to 50 choices) than *metasp* (192 choices). However, this optimization incurs a structural cost for both: the constraint count increases from 3221 to 5058 for *telingo*-λ and from 3508 to 6086 for *metasp*. This behavior, consistent across the benchmark suite, highlights a fundamental key trade-off: both systems achieve a drastic search space reduction at the expense of a significantly inflated grounded representation.

While these findings are promising, a broader empirical and theoretical investigation is necessary to fully characterize this behavior across diverse problem domains. Detailed results for all experiments are available in the *metasp* repository (Hahn *et al*. 2026).

9 Conclusion

We have introduced a meta-programming framework that facilitates the declarative implementation of diverse linear-time temporal logics. By enabling the rapid exploration of alternative logical designs without requiring modifications to the underlying solver, our approach provides a versatile environment suited for both theoretical research and the generation of system blueprints, even beyond the domain of temporal logics.

The framework's efficacy rests on two primary pillars: an extension of the *clingo* theory grammar for temporal expressions and specialized program transformations that insulate nested formulas from the grounder's standard simplifications. From a technical perspective, we have introduced an extended concept of safety that is more permissive than the one established in (Aguado *et al*. 2017). Furthermore, our strategic use of `#external` directives enables the grounding of temporal programs in a single pass, a significant advancement over previous two-pass temporal grounding methods (Aguado *et al*. 2017), bridging the gap between high-level temporal specifications and standard ASP grounding technology. Moreover, we have demonstrated that logic-specific meta-encodings are directly derivable from a logic's satisfaction relation, ensuring semantic clarity and lowering the barrier for proving completeness and correctness, as done by Nemes (2024). Unlike earlier approaches that focus on individual arguments (Balai *et al*. 2013), our typing mechanism targets atomic expressions themselves.

While our current meta-encodings rely on explicit discrete time steps, several promising avenues for future development remain. We aim to explore automata-based computations (Cabalar and Demri 2011; Cuteri *et al*. 2024) and the integration of acyclicity and difference constraints (Behrens *et al*. 2024). Furthermore, we seek to determine how direct temporal encodings (Heljanko and Niemelä 2003; Fionda *et al*. 2024) can be integrated within our declarative meta-programming environment. Ultimately, by bridging the gap

https://doi.org/10.1017/S1471068426100568 Published online by Cambridge University Press

between high-level logical specifications and low-level solver performance, the *metasp* system serves as a flexible and robust platform for the future development of temporal ASP extensions and beyond.

References

Aguado, F., Cabalar, P., Pérez, G., Vidal, C. and Diéguez, M. 2017. Temporal logic programs with variables. *Theory and Practice of Logic Programming* 17, 2, 226–243.

Aguado, F., Cabalar, P., Diéguez, M., Pérez, G., Schaub, T., Schuhmann, A. and Vidal, C. 2023. Linear-time temporal answer set programming. *Theory and Practice of Logic Programming* 23, 1, 2–56. https://doi.org/10.1017/S1471068421000557.

Alur, R. and Henzinger, T. 1992. Logics and models of real time: A survey. In *Real-Time: Theory in Practice*, J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, Eds. Berlin, Heidelberg: Springer-Verlag, 74–106.

Balai, E., Gelfond, M. and Zhang, Y. 2013. Towards answer set programming with sorts. In *Proceedings of the Twelfth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)*, vol. 8148, P. Cabalar and T. Son, Eds. Berlin, Heidelberg: Springer-Verlag, 135–147.

Banbara, M., Kaufmann, B., Ostrowski, M. and Schaub, T. 2017. Clingcon: The next generation. *Theory and Practice of Logic Programming* 17, 4, 408–461. https://doi.org/10.1017/S1471068417000138.

Becker, A., Cabalar, P., Diéguez, M., Hahn, S., Romero, J. and Schaub, T. 2024. Compiling metric temporal answer set programming. In Dodaro et al. (2024), 15–29. https://doi.org/10.1007/978-3-031-74209-5_2.

Becker, A., Cabalar, P., Diéguez, M., Schaub, T. and Schuhmann, A. 2024b. Metric temporal equilibrium logic over timed traces. *Theory and Practice of Logic Programming* 24, 3, 425–452. https://doi.org/10.1017/S1471068424000139.

Behrens, J., Kaminski, R., Schaub, T., Son, T., Švancara, J. and Wanko, P. 2024. Routing and scheduling in answer set programming applied to multi-agent path finding: Preliminary report. abs/2403.12153. https://arxiv.org/abs/2403.12153.

Beiser, A., Hahn, S. and Schaub, T. (2024) ASP-driven user–interaction with clinguin. In *Technical Communications of the Fortieth International Conference on Logic Programming (ICLP'24), EPTCS*, P. Cabalar and T. Swift, Eds. Electronic Proceedings in Theoretical Computer Science.

Cabalar, P. and Demri, S. 2011. Automata-based computation of temporal equilibrium models. In *Proceedings of the Twenty-first International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'11)*, Vol. 7225, Lecture Notes in Computer Science, G. Vidal, Ed. Berlin, Heidelberg: Springer-Verlag, 57–72.

Cabalar, P., Pearce, D. and Valverde, A. 2010. Safety preserving transformations for general answer set programs. In *Proceedings of the Nineteenth International Symposium on Logic Based Program Synthesis and Transformation (LOPSTR'09)*, vol. 6037, Lecture Notes in Computer Science, D. De Schreye, Ed. Berlin, Heidelberg: Springer-Verlag, 132–146.

Cabalar, P., Kaminski, R., Morkisch, P. and Schaub, T. 2019. Telingo = ASP + time. In *Proceedings of the Fifteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'19)*, vol. 11481, Lecture Notes in Artificial Intelligence, M. Balduccini, Y. Lierler and S. Woltran, Eds. Cham: Springer Cham, 256–269. https://doi.org/10.1007/978-3-030-20528-7_19.

Cabalar, P., Diéguez, M., Laferriere, F. and Schaub, T. 2020. Implementing dynamic answer set programming over finite traces. In *Proceedings of the Twenty-Fourth European*

https://doi.org/10.1017/S1471068426100568 Published online by Cambridge University Press

Conference on Artificial Intelligence (ECAI'20), G. De Giacomo, A. Catalá, B. Dilkina, M. Milano, S. Barro, A. Bugarín and J. Lang, Eds. Amsterdam: IOS Press, 656–663. https://doi.org/10.3233/FAIA200151.

Cuteri, A., Mazzotta, G., Peñaloza, R. and Ricca, F. 2024. Automata-based ltl satisfiability checking via ASP. In *Workshop Proceedings of the Twenty-third International Conference of the Italian Association for Artificial Intelligence (AIxIA'24)*, vol. 3883, CEUR Workshop Proceedings, D. Aineto, R. De Benedictis, M. Maratea, M. Mittelmann, G. Monaco, E. Scala, L. Serafini, I. Serina, F. Spegni, E. Tosello, A. Umbrico and M. Vallati, Eds. Aachen, Germany: CEUR-WS.org. https://ceur-ws.org/Vol-3883/paper4_RCRA2.pdf.

De Giacomo, G. and Vardi, M. 2013. Linear temporal logic and linear dynamic logic on finite traces. In *Proceedings of the Twenty-third International Joint Conference on Artificial Intelligence (IJCAI'13)*, F. Rossi, Ed. Menlo Park, California: IJCAI/AAAI Press, 854–860.

Dodaro, C., Gupta, G. and Martinez, M. ed. 2024. Proceedings of the Seventeenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'24), vol. 15245, Lecture Notes in Artificial Intelligence, Springer-Verlag. https://doi.org/10.1007/978-3-031-74209-5.

Fionda, V., Ielo, A. and Ricca, F. 2024. Ltlf2asp: Ltlf bounded satisfiability in ASP. In Dodaro et al. (2024), 373–386. https://doi.org/10.1007/978-3-031-74209-5_28.

Fischer, M. and Ladner, R. 1979. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences* 18, 2, 194–211.

Gebser, M., Kaminski, R. and Schaub, T. 2011. Complex optimization in answer set programming. *Theory and Practice of Logic Programming* 11, 4–5, 821–839.

Hahn, S., Nemes, A., Romero, J. and Schaub, T. 2026. metasp. https://potassco.org/metasp.

Heljanko, K. and Niemelä, I. 2003. Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming* 3, 4–5, 519–550.

Kaminski, R., Romero, J., Schaub, T. and Wanko, P. 2023. How to build your own ASP-based system?! *Theory and Practice of Logic Programming* 23, 1, 299–361. https://doi.org/10.1017/S1471068421000508.

Levesque, H., Reiter, R., Lespérance, Y., Lin, F. and Scherl, R. 1997. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming* 31, 1–3, 59–83. https://doi.org/10.1016/S0743-1066(96)00121-5.

Lifschitz, V. 2019. *Answer Set Programming*. Cham: Springer-Verlag. https://doi.org/10.1007/978-3-030-24658-7.

Nemes, A. 2024. Implementing a temporal answer set solver using metaprogramming techniques, Master's thesis, University of Potsdam.

Pearce, D. 2006. Equilibrium logic. *Annals of Mathematics and Artificial Intelligence* 47, 1–2, 3–41. https://doi.org/10.1007/s10472-006-9028-z.

Pnueli, A. 1977. The temporal logic of programs. In *Proceedings of the Eighteenth Symposium on Foundations of Computer Science (FOCS'77)*. Los Alamitos, California: IEEE Computer Society, 46–57. https://doi.org/10.1109/SFCS.1977.32.

Pratt, V. 1976. Semantical consideration on floyd-hoare logic. In *Proceedings of the Seventeenth Annual Symposium on Foundations of Computer Science (SFCS'76)*. Los Alamitos, California: IEEE Computer Society Press, 109–121.

https://doi.org/10.1017/S1471068426100568 Published online by Cambridge University Press