A Definitional Fragment of Temporal Equilibrium Logic: from Temporal Programs to Compact Automata

Mantas Šimkus[1]

[1]Institute of Logic and Computation, TU Wien, Austria

Abstract

We consider a *definitional fragment* of Temporal Equilibrium Logic (TEL), which allows to capture temporal rules that contain complex (implication-free) temporal formulas in rule bodies. We focus on the setting where formulas are interpreted over *finite traces*. We show that the basic reasoning tasks in the definitional fragment are PSpace-complete, showing a significant drop from the ExpSpace-completeness of the unrestricted case. Our method is based on characterizing stable models in terms of different kinds of *type sequences*, which allow to reason both about the existence and non-existence of stable models. In addition to handling tasks like stable model existence, we also show how the set of all stable models of a theory can be represented as a non-deterministic finite state automaton (NFA). Our method is self-contained in the sense that, in contrast to previous approaches, it does not rely on technical automata-theoretic results as black boxes. Our method can be implemented using SAT solving techniques and a prototype system is available.

Keywords

Temporal Reasoning, Linear Temporal Logic, Equilibrium Logic, Answer Set Programming

1. Introduction

Rule-based languages with default negation are very natural tools for modeling dynamic domains, reasoning about actions and change, automated planning, and performing temporal reasoning more broadly. A key advantage of default negation is its ability to naturally address the *frame problem* by ensuring persistence of facts and inferences over time unless there is justification for change, a feature that is at the core of *Answer Set Programming (ASP)* and related non-monotonic formalisms [1, 2].

However, standard ASP has a significant limitation when it comes to temporal reasoning in situations where the temporal horizon is not known *a priori*. Indeed, standard ASP programs are essentially propositional; rules with variables are handled via *grounding* which, in the context of temporal reasoning, requires us to "hard-code" a bound of the number of time points we consider. Two families of solutions to address this problem have been proposed in the literature. The first one is based on logic programs with function symbols [3, 4]. Indeed, a single unary function symbol in a logic program allows us to model an unbounded timeline. In combination with non-ground ASP rules, this yields a very expressive formalism, which however is immediately undecidable. Thus syntactic restrictions on programs are needed in order to regain decidability and to control the computational complexity [3, 4]. The second family are the works based on *Temporal Equilibrium Logic (TEL)*, which extends *Equilibrium Logic* with temporal modalities [5, 6, 7]. While these languages are decidable, their complexity is in general very high. In particular, reasoning in full TEL is known to be ExpSpace-complete [8]. The same completeness result also applies to *disjunctive BD-programs* considered in [4], when programs are limited to use a single unary function symbol. Several ways to reduce the complexity have been considered: disallowing disjunction [4], or enforcing uni-directional propagation of information along the timeline [3, 5].

In this paper, we contribute to temporal ASP by identifying and studying a *definitional fragment* of TEL. Roughly speaking, we consider temporal programs P that consist of rules of the form $p \leftarrow \varphi$, where p is an atom and φ is an implication-free LTL formula. The rule is universally quantified, with the

24th International Workshop on Nonmonotonic Reasoning, July 17–19, 2026, Lisbon, Portugal
mantas.simkus@tuwien.ac.at (M. Šimkus)
https://www.dbai.tuwien.ac.at/staff/simkus/ (M. Šimkus)
0000-0003-0632-0294 (M. Šimkus)

quantification ranging over all points in time. Intuitively, such rules can be viewed as definitions that assign names to temporal properties of points in time. This setting is related to *nested* logic programs [9] and to *terminologies* in Description Logics under the stable model semantics [10]. We focus on the setting of finite traces, in the spirit of temporal logics on finite traces [11].

The contributions of this paper can be summarized as follows:

- We first identify a *definitional* fragment of TEL that we study in the rest of the paper. We observe that this fragment—when interpreted over finite traces—admits a very simple normal form. This normal form essentially corresponds to *normal BD-programs* and we can infer from [4] that basic reasoning in our fragment is PSPACE-complete. However, the approach in [4] supports infinite traces, and the corresponding automata-based algorithms are more difficult to implement.
- Motivated by the challenge, we present a direct algorithm for checking the existence of a stable model of a temporal program. A particular challenge here is that information can flow in both directions, into the future and into the past. This requires carefully managing the justifications for atoms, ruling out justification cycles that may span even exponentially many time points.
- The basic version of the direct algorithm proves the PSPACE upper bound. This can be seen as a direct proof of the result that can, as mentioned above, alternatively be obtained by an encoding into a normal BD-program.
- We use the direct algorithm as a basis for a more sophisticated algorithm. Specifically, we obtain an incremental procedure that gradually increases the length of traces it considers and alternates between two modes of reasoning: between (a) trying to find a stable model of a given length k, and (b) trying to prove that a stable model does not exist at all.
- The sub-tasks (a) and (b) can be expressed as satisfiability and unsatisfiability checks for propositional logic formulas, i.e. reduced to (UN)SAT. We formally define the incremental procedure that uses (UN)SAT checks as subroutines.
- After dealing with the basic reasoning tasks like consistency, we move to the task of obtaining a small finite representation of the set of all stable models of a temporal program. Specifically, we show how to obtain a "clean" non-deterministic finite state automaton (NFA) that accepts exactly the stable models of the program.
- We have implemented a prototype reasoner that uses the above algorithms. Due to convenience, we use Clingo [12] as a backend, but any other fully-fledged SAT solver could be used instead.

2. Preliminaries

Here we recall *Temporal Equilibrium Logic (TEL)*.

Syntax Let $\mathcal{A}$ denote a countably infinite set of propositional *atoms*. We build *(temporal) formulas* φ using the following grammar:

$$\begin{aligned}\varphi ::= &\mathsf{init} \mid \mathsf{end} \mid \top \mid \bot \mid p \mid \\ &\varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \leftarrow \varphi \mid \neg\varphi \mid \\ &\ominus\, \varphi \mid \boxminus \varphi \mid \varphi\, \mathsf{U}\, \varphi \mid \varphi\, \mathsf{S}\, \varphi \mid \varphi\, \mathsf{R}\, \varphi \mid \varphi\, \mathsf{T}\, \varphi\end{aligned}$$

where $p \in \mathcal{A}$. We use $At(\varphi)$ to denote the atoms that appear in a formula φ.

Semantics An *interpretation* is any set $I \subseteq \mathcal{A}$ of atoms. A *trace* is a sequence of the form

$$\mathcal{I} = I_0 I_1 \cdots I_n$$

where $n \geq 0$ and each I_k is an interpretation. That is, a trace is a finite word over the alphabet $2^{\mathcal{A}}$.

Assume a pair $\mathcal{I} = I_0 I_1 \cdots I_n$ and $\mathcal{J} = J_0 J_1 \cdots J_n$ of traces. We write $\mathcal{I} \subseteq \mathcal{J}$, if $I_k \subseteq J_k$ for all $0 \leq k \leq n$. If $\mathcal{I} \subseteq \mathcal{J}$, then the pair $(\mathcal{I}, \mathcal{J})$ is called an *HT-trace*. We next define a satisfaction relation

'$\models$' that determines the truth value of formulas in the time points $0 \dots n$ of the HT-trace $(\mathcal{I}, \mathcal{J})$. This is done inductively on the structure of formulas as follows:

- $(\mathcal{I}, \mathcal{J}), \ell \models \mathsf{init}$ iff $\ell = 0$
- $(\mathcal{I}, \mathcal{J}), \ell \models \mathsf{end}$ iff $\ell = n$
- $(\mathcal{I}, \mathcal{J}), \ell \models \top$ for all $0 \leq \ell \leq n$
- $(\mathcal{I}, \mathcal{J}), \ell \not\models \bot$ for all $0 \leq \ell \leq n$
- $(\mathcal{I}, \mathcal{J}), \ell \models p$ iff $p \in \mathcal{I}_\ell$
- $(\mathcal{I}, \mathcal{J}), \ell \models \varphi \wedge \psi$ iff $(\mathcal{I}, \mathcal{J}), \ell \models \varphi$ and $(\mathcal{I}, \mathcal{J}), \ell \models \psi$.
- $(\mathcal{I}, \mathcal{J}), \ell \models \varphi \vee \psi$ iff $(\mathcal{I}, \mathcal{J}), \ell \models \varphi$ or $(\mathcal{I}, \mathcal{J}), \ell \models \psi$.
- $(\mathcal{I}, \mathcal{J}), \ell \models \varphi \leftarrow \psi$ iff $(\mathcal{K}, \mathcal{J}), \ell \models \psi$ implies $(\mathcal{K}, \mathcal{J}), \ell \models \varphi$ for both $\mathcal{K} = \mathcal{I}$ and $\mathcal{K} = \mathcal{J}$.
- $(\mathcal{I}, \mathcal{J}), \ell \models \neg\varphi$ iff $(\mathcal{J}, \mathcal{J}), \ell \not\models \varphi$.
- $(\mathcal{I}, \mathcal{J}), \ell \models \oplus \varphi$ iff $\ell < n$ and $(\mathcal{I}, \mathcal{J}), \ell + 1 \models \varphi$.
- $(\mathcal{I}, \mathcal{J}), \ell \models \ominus \varphi$ iff $\ell > 0$ and $(\mathcal{I}, \mathcal{J}), \ell - 1 \models \varphi$.
- $(\mathcal{I}, \mathcal{J}), \ell \models \varphi \,\mathsf{U}\, \psi$ iff there exists $\ell \leq k \leq n$ such that $(\mathcal{I}, \mathcal{J}), k \models \psi$ and $(\mathcal{I}, \mathcal{J}), j \models \varphi$ for all $\ell \leq j < k$.
- $(\mathcal{I}, \mathcal{J}), \ell \models \varphi \,\mathsf{R}\, \psi$ iff one of the following holds: (i) $(\mathcal{I}, \mathcal{J}), k \models \psi$, for all $\ell \leq k \leq n$, or (ii) there exists $\ell \leq k \leq n$ such that $(\mathcal{I}, \mathcal{J}), k \models \varphi$ and $(\mathcal{I}, \mathcal{J}), j \models \varphi$ for all $\ell \leq j < k$.
- $(\mathcal{I}, \mathcal{J}), \ell \models \varphi \,\mathsf{S}\, \psi$ iff there exists $0 \leq k \leq \ell$ such that $(\mathcal{I}, \mathcal{J}), k \models \psi$ and $(\mathcal{I}, \mathcal{J}), j \models \varphi$ for all $k < j \leq \ell$.
- $(\mathcal{I}, \mathcal{J}), \ell \models \varphi \,\mathsf{T}\, \psi$ iff one of the following holds: (i) $(\mathcal{I}, \mathcal{J}), k \models \psi$, for all $0 \leq k \leq \ell$, or (ii) there exists $0 \leq k \leq \ell$ such that $(\mathcal{I}, \mathcal{J}), k \models \varphi$ and $(\mathcal{I}, \mathcal{J}), j \models \varphi$ for all $k < j \leq \ell$.

We say $(\mathcal{I}, \mathcal{J})$ is a *model* of a formula φ if $(\mathcal{I}, \mathcal{J}), 0 \models \varphi$. A trace $\mathcal{J}$ is called a *stable model* of a formula φ, if $(\mathcal{J}, \mathcal{J})$ is a model of φ, and there exists no model $(\mathcal{I}, \mathcal{J})$ of φ such that $\mathcal{I} \subsetneq \mathcal{J}$.

Recall that the following classical temporal modalities can be expressed using the base modalities defined above: 'true at some point in the future' as $\mathsf{F}\,\varphi \stackrel{\text{def}}{=} \top \,\mathsf{U}\, \varphi$, 'always true in the future' as $\mathsf{G}\,\varphi \stackrel{\text{def}}{=} \bot \,\mathsf{R}\, \varphi$, 'true at some point in the past' $\mathsf{P}\,\varphi \stackrel{\text{def}}{=} \top \,\mathsf{S}\, \varphi$, and 'historically true' as $\mathsf{H}\,\varphi \stackrel{\text{def}}{=} \bot \,\mathsf{T}\, \varphi$.

3. Complexity of the Definitional Fragment of TEL

In this section, we introduce the fragment of TEL that we study in the rest of the paper. We then show that the basic reasoning tasks in this fragment are PSPACE-complete. In the following section, we will deal with a more sophisticated incremental solving procedure.

3.1. Definition of the Fragment

In this paper we focus on temporal formulas that are reminiscent of nested logic programs or Description Logic terminologies. We define the following notions.

A *rule* is a formula of the form $p \leftarrow \varphi$, where p is an atom and φ is formula that does not contain '$\leftarrow$'. We say p and φ are the *head* and the *body* of the rule $p \leftarrow \varphi$, respectively. We are specifically interested in formulas of the form:

$$\mathsf{G}\Big(\bigwedge_{0 \leq i \leq n} p_i \leftarrow \varphi_i\Big) \tag{1}$$

where each $p_i \leftarrow \varphi_i$ is a rule. To simplify presentation, any formula of the form (1) can simply be written as a set of rules

$$P = \{p_1 \leftarrow \varphi_1, \dots, p_n \leftarrow \varphi_n\}. \tag{2}$$

Any set P of rules as above is called a *program*. In the following, when we talk about *stable models* of a program P of the form (2), we mean the stable models of the formula (1) that underlies P.

We add some syntactic sugar:

- a "choice rule" $h_1 \mid \ldots \mid h_m \leftarrow \alpha$ stands for the set of all rules $h_k \leftarrow \alpha \wedge \neg \bigvee_{h \in \{h_1,\ldots,h_m\} \setminus \{h_k\}} h$.
- a "constraint" $\bot \leftarrow \alpha$ stands for the rule $f \leftarrow \alpha \wedge \neg f$, where f is a propositional atom that is dedicated to this rule, i.e., f is not allowed to occur anywhere else where the rule is used.

Example 1. *Consider the following scenario: we have a robot that is capable of navigating its environment. The goal is for the robot to exit through a locked door. To unlock the door, the robot must first pick up a key located in a specific area. Consequently, the robot needs to move to the key's location, retrieve the key, then proceed to the door to unlock it and finally exit the room. We will utilize temporal rules to model this domain. Specifically, we want to include a statement for the inertia of holding the key: we want to infer by default that the agent continues to hold the key in the future after having picked it up. This can be modeled using the following temporal program P:*

$$
\begin{aligned}
\mathit{move} \mid \mathit{pick\text{-}key} \mid \mathit{drop\text{-}key} \mid \mathit{unlock} \mid \mathit{exit} &\leftarrow \neg \mathrm{Goal} \\
\mathrm{HasKey} &\leftarrow \ominus\ (\mathit{pick\text{-}key} \vee (\mathrm{HasKey} \wedge \neg \mathit{drop\text{-}key})) \\
\mathrm{Goal} &\leftarrow \ominus\ \mathit{unlock} \\
\bot &\leftarrow \mathit{unlock} \wedge \neg \mathrm{HasKey} \\
\bot &\leftarrow \mathit{drop\text{-}key} \wedge \neg \mathrm{HasKey} \\
\bot &\leftarrow \mathit{pick\text{-}key} \wedge \mathrm{HasKey}
\end{aligned}
$$

*In Figure 1 one can find an automaton that captures all stable models of P (only the "action" predicates are shown in the figure). The automaton was produced by the **ATASP** system, discussed later.*

3.2. Normal Form

We turn our attention to an algorithm for the above fragment. To this end, we can focus on programs in a normal form: we can get rid of complex formulas and focus on simple formulas that resemble standard rules:

Definition 1 (Normal Form)**.** *A program P is in* normal form*, if the following two conditions are satisfied. First, each rule in P is of one of the following forms:*

(F1) $p \leftarrow a_1 \wedge \ldots \wedge a_n \wedge \neg b_1 \wedge \ldots \wedge \neg b_m$,
(F2) $p \leftarrow q \vee q'$
(F3) $p \leftarrow \oplus q$
(F4) $p \leftarrow \ominus q$

where $p, q, q' \in \mathcal{A}$, $\{a_1, \ldots, a_n, b_1, \ldots, b_m\} \subseteq \mathcal{A} \cup \{\mathsf{init}, \mathsf{end}\}$. Second, any atom occurs in the head of at most one rule in P.

Proposition 1. *Any program P can be transformed in polynomial time into a program P' in normal form such that P and P' have the same stable models up to the signature of P.*

Proof. The proof involves two normalization stages. In the first one, we eliminate all nesting in rule bodies by replacing complex sub-formulas using fresh propositional atoms, which in turn are defined using fresh rules. In the second stage, we get rid of the temporal operators U, S, R, and T.

For the first stage, observe the following. Suppose $\varphi[\psi/p]$ denotes the formula that is obtained from φ by replacing each occurrence of the sub-formula ψ in φ by the atom p. Assume a program $P = P_0 \cup \{p \leftarrow \varphi\}$, and suppose ψ is a sub-formula in φ. Consider a new program $P' = P_0 \cup \{p \leftarrow \varphi[\psi/p']\} \cup \{p' \leftarrow \psi\}$. It is not difficult to see that P and P' have the same stable models up to the signature of P. If we iterate such replacements on an initial program P, we can obtain in polynomial time a program that is almost in normal form. In particular, we may still end up with a program that includes rules of the form:

$$p \leftarrow a \,\mathsf{U}\, b \qquad p \leftarrow a \,\mathsf{S}\, b \qquad p \leftarrow a \,\mathsf{R}\, b \qquad p \leftarrow a \,\mathsf{T}\, b \tag{3}$$

where a, b are propositional atoms.

For the second stage, we argue how to eliminate rules of the form (3).

Assume a program $P = P_0 \cup \{p \leftarrow a \,\mathsf{U}\, b\}$. Intuitively, we need to conclude p at every time point i such that there exists j in the future where b holds and until then a hold everywhere. We take a fresh propositional atoms p', p'' and consider the program P' that is obtained from P by replacing $p \leftarrow a \,\mathsf{U}\, b$ with the following rules:

$$p \leftarrow b \vee p' \qquad p' \leftarrow a \wedge p'' \qquad p'' \leftarrow \oplus p$$

It is not difficult to see that P and P' have the same stable models up to the original signature of P. Note that the above 3 rules simply correspond to a normalized version of the rule $p \leftarrow b \vee a \wedge \oplus p$

Assume a program $P = P_0 \cup \{p \leftarrow a \,\mathsf{R}\, b\}$. Intuitively, we need to derive p at every time point i such that b is true at every future time point until a possibly becomes true. We take a fresh atom p' and take the program P' that is obtained from P by replacing $p \leftarrow a \,\mathsf{R}\, b$ with the following rules:

$$p \leftarrow b \wedge p' \qquad p' \leftarrow \oplus p \qquad p \leftarrow a \qquad p \leftarrow b \wedge \mathsf{end}$$

Again, we have P and P' have the same stable model up to the original signature of P. Again note the above 4 rules correspond to a normalized version of a rule $p \leftarrow (b \wedge \oplus p) \vee a \vee (b \wedge \mathsf{end})$.

The cases of S and T are symmetric to the cases of U and R above (we only need to use $\ominus$ and init instead of $\oplus$ and end). □

3.3. Excursion to Encodings of ASP into SAT

The goal of this subsection is to provide intuition behind one of the main components of the algorithm presented later. We will demonstrate how a normal ASP program P can be encoded in polynomial time into a SAT formula φ_P in such a way that the stable models of P correspond to the classical models of φ_P up to the original signature of P. A couple of such encodings are available in the literature exploiting the idea of *level mapping* [13, 14, 15]. Focusing on the original signature of P is essential, as it has been demonstrated in [16] that if φ_P does not incorporate fresh symbols, it may grow exponentially in size relative to P. Readers familiar with this ASP-SAT connection can confidently skip this subsection.

Assume a normal propositional program P with atoms $At(P)$. A *decorated interpretation* for P is a pair $(I, \prec)$, where $I \subseteq At(P)$ and $\prec$ is a strict partial order over I such that the following are satisfied:

(A) For all rules $p \leftarrow a_1 \wedge \ldots \wedge a_n \wedge \neg b_1 \wedge \ldots \wedge \neg b_m$ in P, one of the follwing holds:
- $p \in H$,
- $\{a_1, \ldots, a_n\} \not\subseteq I$, or
- $\{b_1, \ldots, b_m\} \cap I \neq \emptyset$;

(B) For all $p \in I$, there is a rule $p \leftarrow a_1 \wedge \ldots \wedge a_n \wedge \neg b_1 \wedge \ldots \wedge \neg b_m$ in P such that:
- $\{a_1, \ldots, a_n\} \subseteq I$,
- $\{b_1, \ldots, b_m\} \cap I = \emptyset$, and
- $a_k \prec p$ for all $1 \leq k \leq n$;

The condition (A) above requires I to be a classical model of P, while the condition (B) requires each atom to be supported. The last item in (B) together with the fact that $\prec$ is a strict partial order ensures that the support for each atom in I is well-founded. The following can be easily seen:

Proposition 2. *Assume a normal propositional program P. Then I is a stable model of P iff there exists $\prec$ such that $(I, \prec)$ is a decorated interpretation for P.*

Given a program P, we can efficiently produce a SAT formula φ_P whose models correspond to decorated interpretations for P. It uses the original atoms of P, and some fresh atoms. Specifically, (i) for each rule in $\rho \in P$, we take a fresh propositional atom r_ρ, and (ii) for each pair of atoms a, b that appear in P, we take a fresh propositional atom $dep_{a,b}$. Intuitively, $dep_{a,b}$ means "a depends on b", and will correspond to $b \prec a$ in a desired strict partial order $\prec$. Then the formula φ_P is the conjuntion of the following subformulas:

(S1) ρ for every rule $\rho \in P$,

(S2) $(dep_{a,b} \wedge dep_{b,c} \rightarrow dep_{a,c}) \wedge \neg dep_{a,a}$ for all propositional atoms a, b, c that appear in P,

(S3) $h \rightarrow r_{\rho^1} \vee \ldots \vee r_{\rho^k}$ for each atom h of P, where $\{\rho^1, \ldots, \rho^k\}$ is the set of all rules of P with h in the head,

(S4) $r_\rho \rightarrow a_1 \wedge dep_{a_1,p} \wedge \cdots \wedge a_n \wedge dep_{a_n,p} \wedge \neg b_1 \wedge \cdots \wedge \neg b_m$ for all rules $\rho = p \leftarrow a_1 \wedge \cdots \wedge a_n \wedge \neg b_1 \wedge \cdots \wedge \neg b_m$ in P.

The four subformulas require a model I of φ_P: (S1) to be a classical model of P, (S2) to correctly encode a strict partial order, (S3) for every atom of I to select a supporting rule, and (S4) to ensure that the body of each chosen supporting rule is satisfied and that the atom dependencies are asserted. Based on Proposition 2 and the construction of φ_P, we can conclude that φ_P captures all stable models of P up to the signature of P. Specifically, we have:

Proposition 3. *Assume a normal propositional program P and the formula φ_P computed for P. We have that I is a stable model of P if and only if φ_P has a classical model J with $I = J \cap At(P)$.*

It is important to note that the formula φ_P generated by the above method may exhibit cubic growth relative to the size of the input program P. This could pose challenges when dealing with larger programs. For more advanced methods, refer to [13, 14, 15].

3.4. Characterization of Stable Models and PSpace-completeness

In the rest of the paper, we focus solely on temporal programs in normal form, and thus "program" always means "program in normal form", unless explicitly specified otherwise.

Our goal is now to characterize stable models of a program P via *sequences of types*. Intuitively, a *type* here is a description of a time point, which consists of two parts: (a) the atoms that are true at the time point, and (b) the local dependencies related to those atoms. The challenge we have to handle is that we have temporal modalities $\ominus$ and $\oplus$, which means that atoms may depend on atoms in the past and in the future.

Before we introduce types and type sequences, we introduce a characterization of stable models that is rather direct adaptation of the idea of seeing stable models as classical models which additionally enjoy a *level mapping*. A level mapping prohibits cyclic justification of atoms. In our setting, we use classical models that can in addition be annotated with a well-founded *derivation relation*.

Definition 2. *Assume a trace $\mathcal{I} = I_0 \cdots I_n$. We let*

$$\bar{\mathcal{I}} = \{(p, i) \mid p \in I_i, 0 \leq i \leq n, p \neq \mathsf{init}, p \neq \mathsf{end}\}$$

A derivation relation *for $\mathcal{I}$ w.r.t. a program P is a binary relation $\rightsquigarrow \subseteq \bar{\mathcal{I}} \times \bar{\mathcal{I}}$ such that one of the following holds for all $(p, j) \in \bar{\mathcal{I}}$:*

1. *There is $p \leftarrow a_1 \wedge \ldots \wedge a_n \wedge \neg b_1 \wedge \ldots \wedge \neg b_m \in P$ such that*
   a) *$\{b_1, \ldots, b_m\} \cap I_j = \emptyset$, and*
   b) *$(p, j) \rightsquigarrow (a_k, j)$, for all $1 \leq k \leq n$ with $a_k \notin \{\mathsf{init}, \mathsf{end}\}$.*
2. *There is a rule $p \leftarrow a_1 \vee a_2$ in P such that $(p, j) \rightsquigarrow (a_1, j)$ or $(p, j) \rightsquigarrow (a_1, j)$.*
3. *We have $j < n$ and there is a rule $p \leftarrow \oplus q$ in P such that $(p, j) \rightsquigarrow (q, j + 1)$.*
4. *We have $j > 0$ and there is a rule $p \leftarrow \ominus q$ in P such that $(p, j) \rightsquigarrow (q, j - 1)$.*

We say $\rightsquigarrow$ is well-founded *if it does not contain an infinite chain $(p_0, i_0) \rightsquigarrow (p_1, i_1) \rightsquigarrow (p_2, i_2) \rightsquigarrow \ldots$*

Note that, since $\bar{\mathcal{I}}$ is finite, $\rightsquigarrow$ is well-founded iff $\rightsquigarrow$ does not have a cycle, i.e., there is no $(p_0, i_0) \in \bar{\mathcal{I}}$ such that $(p_0, i_0) \rightsquigarrow (p_1, i_1) \rightsquigarrow \ldots \rightsquigarrow (p_0, i_0)$.

Derivation relations properly characterize stable model of programs.

Theorem 1. *Assume a program P and a trace $\mathcal{I}$. Then $\mathcal{I}$ is a stable model of P iff the following are satisfied:*

(A) *$(\mathcal{I}, \mathcal{I})$ is a model of P, and*
(B) *there exists a well-founded derivation relation $\rightsquigarrow$ for $\mathcal{I}$ w.r.t. P.*

We next develop a method for building traces that can be "decorated" with a well-founded derivation relation, and thus for building stable models. Specifically, we will be interested in sequences of types where pairs of successor types satisfy certain compatibility conditions: such type sequence encode a classical model enriched with a well-founded derivation relation. We will also see that any stable model of a program can be decomposed into a sequence of compatible types.

Definition 3. *A* type *for a program P is any pair (I, D), where $I \subseteq At(P)$ and $D \subseteq I \times \{-1, 0, +1\} \times At(P)$, such that the following conditions are satisfied:*

(a) *I satisfies every rule of the form (F1) in P.*
(b) *We have $(p, 0, r) \in D$ whenever $(p, 0, q), (q, 0, r) \in D$.*
(c) *There exists no $(p, 0, p) \in D$.*
(d) *For all $p \in I$, one of the following holds:*
  (i) *There is a rule $p \leftarrow a_1 \wedge \ldots \wedge a_n \wedge \neg b_1 \wedge \ldots \wedge \neg b_m$ in P such that*
    - *$\{a_1, \ldots, a_n\} \subseteq I$,*
    - *$\{b_1, \ldots, b_m\} \cap I = \emptyset$, and*
    - *$(p, 0, a_k) \in D$ for all $1 \leq k \leq n$ with $a_k \notin \{\mathsf{init}, \mathsf{end}\}$;*
  (ii) *There is a rule $p \leftarrow a_1 \vee a_2$ in P such that*
    - *$(p, 0, a_1) \in D$ and $a_1 \in I$, or*
    - *$(p, 0, a_2) \in D$ and $a_2 \in I$.*
  (iii) *There is a rule $p \leftarrow \oplus q$ in P such that $(p, +1, q) \in D$;*
  (iv) *There is a rule $p \leftarrow \ominus q$ in P such that $(p, -1, q) \in D$.*

A type (I, D) with $\mathsf{init} \in I$ (resp., $\mathsf{end} \in I$) is called a starting *(resp.,*final*) type. We use $\mathsf{types}(P)$ to denote the set of types for P.*

Next we see when a pair of types is 'compatible'.

Definition 4. *A type (I', D') is a* good successor *for a type (I, D), if the following are satisfied:*

(a) *if $(p, +1, q) \in D$, then $q \in I'$;*
(b) *if $(p, -1, q) \in D'$, then $q \in I$;*
(c) *if $(p_1, -1, p_2) \in D'$ and $(p_2, 0, p_3), (p_3, +1, p_4) \in D$, then $(p_1, 0, p_4) \in D'$;*
(d) *if $p \leftarrow \ominus q \in P$ and $q \in I$, then $p \in I'$.*
(e) *if $p \leftarrow \oplus q \in P$ and $q \in I'$, then $p \in I$.*

Definition 5. *Assume a sequence $S = (I_0, D_0), \ldots, (I_n, D_n)$ of types for P. We say S is a* good *type sequence for P, if*

- *$\mathsf{init} \in I_0$ and $\mathsf{init} \notin I_i$ for all $0 < i \leq n$,*
- *$\mathsf{end} \in I_n$ and $\mathsf{end} \notin I_i$ for all $0 \leq i < n$, and*
- *(I_i, D_i) is a good successor of (I_{i-1}, D_{i-1}), for all $0 < i \leq n$.*

Theorem 2. *If $(I_0, D_0), \ldots, (I_n, D_n)$ is a good type sequence for a program P, then $I_0 \cdots I_n$ is a stable model of P. If $I_0 \cdots I_n$ is a stable model of P, then there exist $D_0, \ldots, D_n$ such that $(I_0, D_0), \ldots, (I_n, D_n)$ is a good type sequence for P.*

Proof. For the first claim, assume $(I_0, D_0), \ldots, (I_n, D_n)$ is a good type sequence for a program P. We need to see that $\mathcal{I} = I_0 \cdots I_n$ is a stable model of P, for which, based on Theorem 1, it suffices to see that (i) $(\mathcal{I}, \mathcal{I})$ is a model of P, and that (ii) there exists a well-founded derivation relation $\rightsquigarrow$ for $\mathcal{I}$ w.r.t. P.

For (i), we need to see that all rules of the forms (F1-F4) from P are satisfied. The rules of the forms (F1) and (F2) are satisfied because of the condition (a) in Definition 3. The rules of the form (F3) and (F4) are satisfied because of the conditions (d) and (e) in Definition 4.

For the claim (ii), we construct a proper derivation relation $\rightsquigarrow$ over $\bar{\mathcal{I}}$, which is obtained by taking, for all $0 \leq i \leq n$, the following pairs:

- $((p, i), (q, i))$ for all $(p, 0, q) \in D_i$,
- $((p, i), (q, i-1))$ for all $(p, -1, q) \in D_i$, and
- $((p, i), (q, i+1))$ for all $(p, +1, q) \in D_i$.

It only remains to see that $\rightsquigarrow$ does not contain a cycle. This follows from the definition of types, specifically points (b) and (c) of Definition 3, in combination with condition (c) in Definition 4. Intuitively, if $\rightsquigarrow$ had a cycle, then (c) in Definition 4 would ensure there is a cycle that is localized to a particular time point. Such cycle is the disallowed by (b) and (c) of Definition 3.

For the second claim of the theorem, assume a stable model $\mathcal{I} = I_0 \cdots I_n$ of P. Based on Theorem 1, we have that (i) $(\mathcal{I}, \mathcal{I})$ is a model of P, and that (ii) there exists a well-founded derivation relation $\rightsquigarrow$ for $\mathcal{I}$ w.r.t. P. We need to build $D_0, \ldots, D_n$ such that $(I_0, D_0), \ldots, (I_n, D_n)$ is a good type sequence for P. We can w.l.o.g. assume that $(p, i) \rightsquigarrow (q, j)$ implies $j - i \in \{-1, 0, +1\}$. We define each D_i are follows. We let $D_i = \{(p, j-i, q) \mid (p, i) \rightsquigarrow (q, j)\}$. It can be easily verified that resulting type sequence is a good type sequence for P. □

Theorem 3. *Deciding stable model existence for temporal programs PSPACE-complete.*

Proof. Assume a program P. Every type (I, D) for P is of polynomial size in the size of P. Checking whether (I', D') is a good successor type for (I, D') requires only polynomial time. Thus we can check the existence of good type sequence for P by non-deterministically guessing a sequence of types, keeping in memory only a pair of types and counter that counts to at most $|\mathsf{types}(P)|$. The counter requires only polynomially many bits in the size of P. This procedure can be implemented to run in polynomial space.

The matching PSPACE lower bound follows easily, e.g., from [4]. □

Several other reasoning tasks can be reduced to the stable model existence problem: e.g. we may be interested in *cautious/brave* entailment of an (implication-free) temporal formula at the initial time point. Specifically, an implication-free formula φ is cautiously (resp. bravely) entailed by a program P at the initial time point iff $P \cup \{\leftarrow \mathsf{init} \wedge \varphi\}$ has no stable model (resp. $P \cup \{\leftarrow \mathsf{init} \wedge \mathit{not}\, \varphi\}$ has a stable model).

4. Efficient Reasoning via Incremental Solving

In this section, we describe a concrete procedure for reasoning in temporal programs. Specifically, we first describe a procedure for checking the existence of a stable model of a program, and then see how it can be used to obtain a finite representation of all stable models a program as an NFA. To check the existence of stable models, our procedure gradually increases the length of candidate stable models, and specifically employs SAT solving to reason about type sequences. This procedure is the basis for our implementation that we describe in the next section.

4.1. Reasoning using Iterative SAT and UNSAT Checks

As the first step, we show how to build, given a program P and an integer k, a propositional formula φ such that φ is satisfiable iff P has a stable model of length k. For this we directly use Theorem 2: the formula φ encodes the existence of a good type sequence for P of length k. Given efficient SAT solvers, this approach has the potential to quickly find stable models of small length when they exist. Afterwards, we shall see a trick that uses UNSAT to identify situations that witness non-existence of stable models. The overall procedure is then to gradually increase k and run the two checks to either find a stable model of length k or prove that a stable model of length $\geq k$ does not exist.

Definition 6. *Assume a program P and an integer $k \geq 0$. For all $0 \leq j \leq k$ and each $\lambda \in At(P) \cup \{\mathsf{init}, \mathsf{end}\}$, we use a fresh propositional variable λ^j. We define formulas start^k and final^k as follows:*

$$\mathsf{start}^k := \mathsf{init}^0 \wedge \bigwedge_{0<j\leq k} \neg \mathsf{init}^j$$

$$\mathsf{final}^k := \mathsf{end}^k \wedge \bigwedge_{0\leq j<k} \neg \mathsf{end}^j$$

Moreover, for all $0 \leq j \leq k$ and all pairs $p, q \in At(P)$, we use a fresh propositional variables $\mathsf{dep}^i_{(p,0,q)}$, $\mathsf{dep}^i_{(p,-1,q)}$, and $\mathsf{dep}^i_{(p,+1,q)}$. We construct a formula succ^k_P as the conjunction of the following formulas below:

- *for all $p, q, r \in At(P)$ and $0 \leq i \leq k$, we include*

$$\mathsf{dep}^i_{(p,0,q)} \wedge \mathsf{dep}^i_{(q,0,r)} \rightarrow \mathsf{dep}^i_{(p,0,r)} \tag{4}$$

- *for all $p \in At(P)$ and $0 \leq i \leq k$, we include $\neg\mathsf{dep}^i_{(p,0,p)}$*
- *for all rules $p \leftarrow a_1, \ldots, a_n, \neg b_1, \ldots, \neg b_m$ of type (F1) in P and $0 \leq i \leq k$, we include*

$$p^i \equiv a^i_1 \wedge \cdots \wedge a^i_n \wedge \neg b^i_1 \wedge \cdots \wedge \neg b^i_m \tag{5}$$

$$p^i \rightarrow \bigwedge_{1\leq j\leq n, a_j \notin \{\mathsf{init},\mathsf{end}\}} \mathsf{dep}^i_{(p,0,a_j)} \tag{6}$$

- *for all rules $p \leftarrow p_1 \vee p_2$ of P and $0 \leq i \leq k$, we include*

$$p^i_1 \vee p^i_2 \rightarrow p^i \tag{7}$$

$$p^i \rightarrow (\mathsf{dep}^i_{(p,0,p_1)} \wedge p^i_1) \vee (\mathsf{dep}^i_{(p,0,p_2)} \wedge p^i_2) \tag{8}$$

- *for all rules $p \leftarrow \ominus q$ of P and $0 < i \leq k$, we include*

$$(q^{i-1} \leftrightarrow p^i) \wedge (p^i \rightarrow \mathsf{dep}^i_{(p,-1,q)}) \tag{9}$$

- *for all rules $p \leftarrow \oplus q$ of P and $0 \leq i < k$, we include*

$$(q^{i+1} \leftrightarrow p^i) \wedge (p^i \rightarrow \mathsf{dep}^i_{(p,+1,q)}) \tag{10}$$

Since the above propositional formulas directly mirror the conditions for good type sequences, we immediately get the following:

Proposition 4. *A program P has a stable model of length k iff the formula $\mathsf{start}^k \wedge \mathsf{succ}^k_P \wedge \mathsf{final}^k$ is satisfiable.*

Our next goal is find a way to spot when a program has no stable model at all. For this, we consider *progress witnesses*, which differ from good type sequences in two ways: (i) they do not need to contain the end marker in the final type of the sequence, and (ii) the sequence is not allowed to repeat types.

Algorithm 1 Incremental Procedure for Consistency Testing

Require: program P
Ensure: *true* iff P has a stable model

```
k ← 0
while k < |types(P)| do
    if start^k ∧ succ_P^k ∧ final^k is satisfiable
        then return true
    if start^k ∧ succ_P^k ∧ nonrep_P^k is unsatisfiable
        then return false
    k ← k + 1
end while
return false
```

Definition 7. *Assume a sequence $S = (I_0, D_0), \ldots, (I_n, D_n)$ of types for a program P. We say S is a* progress witness *for P, if*

- $\mathsf{init} \in I_0$,
- *(I_i, D_i) is a good successor of (I_{i-1}, D_{i-1}), for all $0 < i \leq n$, and*
- *every type in S appears not more than once.*

Proposition 5. *Assume a program P and an integer k. Suppose P has no stable model of length $< k$, and there exists no progress witness for P of length k. Then P has no stable model at all.*

Proof. Assume P and k as in the claim. Towards a contradiction, assume P has a stable model $\mathcal{I}$. We will argue that this implies the existence of a progress witness for P of length k. Indeed, since $\mathcal{I}$ is a stable model, P has a good type sequence S. Assume a type (I, D) that appears in S first in position i and last in position j with $j > i$ (thus, the type appears at least twice in S). Consider the type sequence $S' = (I_0, D_0), \ldots, (I_i, D_i), (I_{j+1}, D_{j+1}), \ldots, (I_n, D_n)$. Observe that S' is a good type sequence for P that contains exactly one occurrence of (I, D). Thus by repeating the above elimination of type repetitions, we can obtain a good type sequence S'' for P that does not repeat any types, i.e. the length of S'' is exactly the number of types that appear in S. Since P has no stable model of length $< k$, it must be the case that $|S''| \geq k$. Then the initial prefix of S'' consisting of the first k types is a progress witness for P of length k. □

We can encode the task of checking the existence of a progress witness into SAT. The encoding is based on the previous formulas; we only need machinery to prohibit type repetition in type sequences.

Definition 8. *Assume a program P and an integer $k \geq 0$. Let Θ^i denote the set of all super-scripted propositions θ^i introduced in Definition 6. We define the following formula:*

$$\mathsf{nonrep}_P^k := \bigwedge_{0 \leq i < j \leq k} \bigvee_{\theta^i \in \Theta^i, \theta^j \in \Theta^j} \neg(\theta^i \leftrightarrow \theta^j)$$

Proposition 6. *A program P has a progress witness of length k iff the formula $\mathsf{start}^k \wedge \mathsf{succ}_P^k \wedge \mathsf{nonrep}_P^k$ is satisfiable.*

All the above can be put together into a procedure presented in Algorithm 1. Its soundness and completeness follows directly from Proposition 4 and 5, and the fact that the search for stable models can be limited to type sequences whose length is bounded by the number of types for a input program.

Theorem 4. *Algorithm 1 is a sound and complete procedure for checking the existence of a stable model for a program P.*

Note that the procedure in Algorithm 1 is not a polynomial space procedure because k may grow exponentially in the size of the input program. However, we are certain that is not a major problem. We believe that in most practical cases stable models will not be extremely long, and in cases when a stable model does not exist this will be proven quickly by the nonexistence of short progress witnesses.

4.2. Finite Representation of Stable Models Using an NFA

Our next goal is to obtain a finite representation of all stable models of a given program. For this we use an NFAs. Specifically, for a given program P we will obtain an NFA A_P whose language is the exactly the stable models of P. We believe that have an NFA representation can be used in diverse downstream applications, e.g., for supporting agent's planning tasks by making the agent aware of the dynamics of the system. The basic expressiveness of NFAs to represent stable models of temporal program is not surprising; e.g., for this one may use the approach in [4]. However, we need be careful to be careful since naive approaches may easily lead to automata of exponential size, which then renders the NFA hardly usable. Our approach here is to exploit the the method from the previous subsection to obtain a structure that precisely captures all good type sequences for a program P. This structure can be be directly converted into an NFA.

Definition 9. *A* trace representation *for a program P is a pair $\mathcal{R} = (\mathcal{B}, \mathcal{S})$, where $\mathcal{B}$ is the set of starting types for P, and $\mathcal{S}$ is a binary relation over types such that $(\tau, \tau') \in \mathcal{S}$ implies that τ' is a good successor of τ w.r.t. P. We define $\mathsf{paths}(\mathcal{S})$ as the smallest set of words over types such that:*

(a) $\mathcal{B} \subseteq \mathsf{paths}(\mathcal{R})$*, and*
(b) *if $\tau_1 \cdots \tau_k \in \mathsf{paths}(\mathcal{R})$ and $(\tau_k, \tau') \in \mathcal{S}$, then also $\tau_1 \cdots \tau_k \cdot \tau' \in \mathsf{paths}(\mathcal{R})$.*

A sequence $\tau_1 \cdots \tau_k \in \mathsf{paths}(\mathcal{R})$ is called terminal*, if there is no τ' such that $\tau_1 \cdots \tau_k \cdot \tau' \in \mathsf{paths}(\mathcal{R})$. We say $\mathcal{R}$ is* redundancy-free*, if the following are satisfied:*

(a) *for all terminal sequences $\tau_1 \cdots \tau_k \in \mathsf{paths}(\mathcal{R})$ we have that τ_k is a final type for P, and*
(b) *for all $(\tau, \tau') \in \mathcal{S}$, we have τ appears in some sequence from $\mathsf{paths}(\mathcal{R})$.*

Let $\mathsf{traces}(\mathcal{R})$ consist of $I_1 \cdots I_k$ for each terminal $(I_1, D_1) \cdots (I_k, D_k) \in \mathsf{paths}(\mathcal{R})$. We say $\mathcal{R}$ is a proper *trace representation for P, if $\mathcal{R}$ is* redundancy-free *and $\mathcal{I} \in \mathsf{traces}(\mathcal{R})$ for all stable model $\mathcal{I}$ of P.*

Assume a trace representation $\mathcal{R} = (\mathcal{B}, \mathcal{S})$ for a program P. Intuitively, starting from types in $\mathcal{B}$ we can build sequences of types by following the successor relation $\mathcal{S}$. We know that any such sequence that ends with a final type corresponds to one stable model of P. If we assume $\mathcal{R}$ is redundancy-free, then at any point in construction of the sequence we can always find at least one successor type, unless we have already reached a final type. Overall, $\mathcal{R}$ being redundancy-free means that all types in $\mathcal{B}$ and all tuples in $\mathcal{S}$ participate in the construction of some stable model of P. Together with condition that $\mathcal{R}$ includes all stable models of P, a proper trace representation captures exactly the stable models of P.

Proposition 7. *For any program P, there exists a proper trace representation $\mathcal{R}$. Moreover, $\mathcal{R}$ is unique and $\mathsf{traces}(\mathcal{R})$ is exactly the set of stable models of P.*

Proof. Let T be the set of all good type sequences for P. Let $\mathcal{B} = \{\tau_0 \mid \tau_0 \cdot \tau_1 \cdots \tau_n \in T\}$ and $\mathcal{S} = \{(\tau_i, \tau_{i+1}) \mid \tau_0 \cdots \tau_i \cdot \tau_{i+1} \cdots \tau_n \in T\}$. It can be verified that $\mathcal{R} = (\mathcal{B}, \mathcal{S})$ is a proper trace representation for P, that $\mathcal{R}$ is the only proper trace representation for P, and $\mathsf{traces}(\mathcal{R})$ is exactly the set of stable models of P. □

We next show how to compute the proper trace representation $\mathcal{R}$ for a program P. To this end, for a binary relation R over types and a type τ, let $R(\tau) = \{\tau' \mid (\tau, \tau') \in R\}$. The procedure for computing $\mathcal{R}$ is presented in Algorithm 2. It uses a subroutine $\mathsf{check}_P(\tau_1, T)$ that takes as input a program P, a type τ_1, and a set T of types. The procedure $\mathsf{check}_P(\tau_1, T)$ decides if there is a good type sequence of the form $\tau_1 \cdot \tau_2 \cdots \tau_k$ such that $\tau_2 \notin T$. If such a sequence exists, the procedure returns τ_2. If it does not exist, it returns *nil*. We can implement $\mathsf{check}_P(\tau_1, T)$ using a small adaptation of Algorithm 1, and thus employ a SAT solver.

To understand Algorithm 1, assume a program P. The procedure starts by computing the set $\mathcal{B}$ of all types that contain both init and end in the type, and thus captures all stable models of length 1. It then computes a set $\mathcal{A}$ as the set of all initial types starting from which there exists some good type sequence for P. Adding $\mathcal{A}$ to $\mathcal{B}$ makes sure that $\mathcal{B}$ consists of exactly the types that correspond to the

Algorithm 2 Computing a Proper Trace Representation

Require: program P
Ensure: $\mathcal{R} = (\mathcal{B}, \mathcal{S})$ a proper trace representation for P
1: Let $\mathcal{B}$ be the set of all types for P that are both initial and final
2: Let $\mathcal{A}$ be the set of all initial types τ such that $\mathsf{check}_P(\tau, \emptyset) \neq \emptyset$
3: Add $\mathcal{A}$ to $\mathcal{B}$
4: **while** exists $\tau \in \mathcal{A}$ such that $\mathsf{check}_P(\tau, \mathcal{S}(\tau)) \neq nil$ **do**
5: Let $\tau' = \mathsf{check}_P(\tau, \mathcal{S}(\tau))$
6: Add (τ, τ') to $\mathcal{S}$
7: Add τ' to $\mathcal{A}$
8: **end while**
9: **return** $\mathcal{R} = (\mathcal{B}, \mathcal{S})$

first types of all stable models of P. The types in $\mathcal{A}$ are the "active" types; these are the types for which we need add all relevant successor types. This is done in the "while" loop: if a new successor τ' for an active type τ is found, then this is recorded by adding (τ, τ') to the successor relation and adding to τ' to the set of active types. Overall, one can see the following:

Theorem 5. *Algorithm 2 returns a proper trace representation for a program P.*

A proper trace representation $\mathcal{R} = (\mathcal{B}, \mathcal{S})$ captures good type sequences, but here types include the extra information related atom dependencies. We can project away that information and capture plain stable models of a program. For this, we can use standard NFAs. An NFA is given as a tuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q is a set of states, Σ is an alphabet, $q_0 \in Q$ is the initial states, $F \subseteq Q$ is a set of final states, and δ is a transition relation $\delta \subseteq Q \times \Sigma \cup \times Q$. The language accepted by A is denoted $L(A)$.

Definition 10. *Assume a program P and let $\mathcal{R} = (\mathcal{B}, \mathcal{S})$ be a proper trace representation for P. We construct an NFA $A_P = (Q, \Sigma, \delta, q_0, F)$ as follows:*

(a) $Q = \{\tau \mid \tau$ *appears in* $\mathcal{R}\} \cup \{q_0\}$,
(b) Σ *is set of all subsets of* $At(P) \cup \{\mathsf{init}, \mathsf{end}\}$,
(c) F *is the set of all final types in* Q, *and*
(d) δ *consists of the following:*
  (i) $((I, D), I', (I', D'))$ *for all* $((I, D), (I', D')) \in \mathcal{S}$,
  (ii) $(q_0, I, (I, D))$ *for all initial types in* Q.

The automaton A_P is the desired representation of the stable models of a program P.

Proposition 8. *If P is a program, then $L(A_P)$ is the set of stable models of P.*

Proof. Assume $I_0 \cdots I_n$ is a stable model of P. By Theorem 2, there exist $D_0, \ldots, D_n$ such that $(I_0, D_0), \ldots, (I_n, D_n)$ is a good type sequence for P. Observe that $q_0, (I_0, D_0), \ldots, (I_n, D_n)$ corresponds to an accepting run of A_P on $I_0 \cdots I_n$, and thus $I_0 \cdots I_n \in L(A_P)$.

Assume a word $I_0 \cdots I_n \in L(A_P)$ that is witnessed by the run $q_0, (I_0, D_0), \ldots, (I_n, D_n)$. By the construction of the automaton, $(I_0, D_0), \ldots, (I_n, D_n)$ is a good type sequence for P and thus $I_0 \cdots I_n$ is a stable model of P. □

5. Implementation

We have implemented a prototype system **ATASP** (**A**utomata for **T**emporal **ASP**) that implements the reasoning method described in this paper[1]. Specifically, given a temporal program P as input, **ATASP**

[1] https://github.com/mantassimkus/temporal-asp

```
hold_key :- (-1) (do_pick_key | hold_key & not do_drop_key).
do_move :- not do_pick_key & not do_drop_key & not do_unlock & not goal.
do_pick_key :- not do_move & not do_drop_key & not do_unlock & not goal.
do_drop_key :- not do_pick_key & not do_move & not do_unlock & not goal.
do_unlock :- not do_pick_key & not do_drop_key & not do_move & not goal.
do_exit:- (-1) do_unlock.
:- do_unlock & not hold_key.
:- do_drop_key & not hold_key.
:- do_pick_key & hold_key.
goal :- (-1) (do_unlock).
:- goal & not end.
:- end & not goal.
```

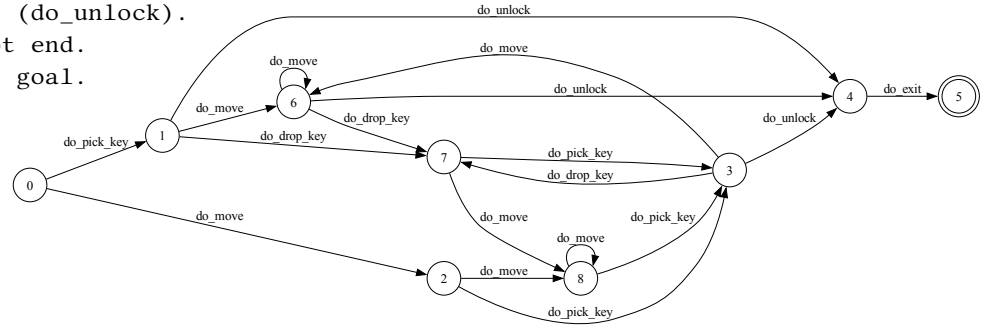

Figure 1: A temporal program in **ATASP** syntax capturing Example 1; the NFA resulting from the program obtained by **ATASP**.

computes the NFA A_P and visualizes it using the *Graphviz* library [17]. The system is implemented in Python and it directly uses the algorithms described above. For checking (un)satisfiability of constructed propositional formulas it uses the ASP solver Clingo [12], but in principle a dedicated SAT solver could be used instead. To get an impression of the system, see Figure 1 for a formalization of Example 1.

6. Conclusion

In this paper, we studied a definitional fragment of Temporal Equilibrium Logic, which allows a restricted usage of the implication symbol in formulas. The language is still flexible enough to allow temporal programs that intuitively capture terminologies of temporal situations, i.e., it supports giving names to time points that satisfy a complex (implication-free) temporal formula. We have shown that basic reasoning tasks like checking the existence of a stable model of a temporal program are PSpace-complete. However, the employed technique is more interesting than the complexity result itself. In our fragment, information can flow in two directions (from the past to the future, and vice versa), which makes reasoning challenging. We have addressed this problem by means of carefully designed notions of types and sequences of types. This characterization opened the way to an incremental solving procedure that promises to not only quickly find stable models when they exist, but also to quickly return a negative answer when a stable model does not exist. The incremental procedure was then lifted to be able to compute the finite representation of stable models using NFAs.

There are some directions for future work. First, we would like to extend the fragment to the first-order setting, where variables would be allowed in temporal formulas and programs. In terms of theory, this extension is not difficult, but the actual implementation in **ATASP** would require significant efforts. Another direction is to extend our definitional fragment to include more formulas without increasing the computational complexity. To this end, we plan to study a *Harrop* fragment of TEL [18].

Acknowledgments

This work was supported in part by the Austrian Science Fund (FWF) projects 10.55776/COE12 and 10.55776/PIN8884924.

Declaration on Generative AI

During the preparation of this work, the author(s) used ChatGPT (GPT-5.3) in order to: (a) grammar and spelling check, (b) paraphrase and reword. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming, in: R. A. Kowalski, K. A. Bowen (Eds.), Proc. of ICLP 1988, MIT Press, 1988, pp. 1070–1080.

[2] G. Brewka, T. Eiter, M. Truszczyński, Answer set programming at a glance, Commun. ACM 54 (2011) 92–103. URL: https://doi.org/10.1145/2043174.2043195. doi:10.1145/2043174.2043195.

[3] T. Eiter, M. Šimkus, FDNC: decidable nonmonotonic disjunctive logic programs with function symbols, ACM Trans. Comput. Log. 11 (2010) 14:1–14:50. doi:10.1145/1656242.1656249.

[4] T. Eiter, M. Šimkus, Bidirectional answer set programs with function symbols, in: C. Boutilier (Ed.), Proc. of IJCAI 2009, 2009, pp. 765–771. URL: http://ijcai.org/Proceedings/09/Papers/132.pdf.

[5] D. Pearce, Equilibrium logic, Annals of Mathematics and Artificial Intelligence 47 (2006) 3–41. URL: https://doi.org/10.1007/s10472-006-9028-z. doi:10.1007/s10472-006-9028-z.

[6] F. Aguado, P. Cabalar, M. Diéguez, G. Pérez, T. Schaub, A. Schuhmann, C. Vidal, Linear-time temporal answer set programming, Theory Pract. Log. Program. 23 (2023) 2–56. URL: https://doi.org/10.1017/S1471068421000557. doi:10.1017/S1471068421000557.

[7] A. Bosser, P. Cabalar, M. Diéguez, T. Schaub, Introducing temporal stable models for linear dynamic logic, in: M. Thielscher, F. Toni, F. Wolter (Eds.), Proc. of KR, AAAI Press, 2018, pp. 12–21. URL: https://aaai.org/ocs/index.php/KR/KR18/paper/view/18047.

[8] L. Bozzelli, D. Pearce, On the complexity of temporal equilibrium logic, in: Proc. of LICS 2015, LICS '15, IEEE Computer Society, USA, 2015, p. 645–656. doi:10.1109/LICS.2015.65.

[9] V. Lifschitz, L. R. Tang, H. Turner, Nested expressions in logic programs, Ann. Math. Artif. Intell. 25 (1999) 369–389. URL: https://doi.org/10.1023/A:1018978005636. doi:10.1023/A:1018978005636.

[10] F. D. Stefano, M. Šimkus, Stable model semantics for description logic terminologies, in: M. J. Wooldridge, J. G. Dy, S. Natarajan (Eds.), Proc. of AAAI 2024, AAAI Press, 2024, pp. 10484–10492. URL: https://doi.org/10.1609/aaai.v38i9.28917. doi:10.1609/AAAI.V38I9.28917.

[11] G. D. Giacomo, M. Y. Vardi, Linear temporal logic and linear dynamic logic on finite traces, in: F. Rossi (Ed.), Proc. of IJCAI 2013, IJCAI/AAAI, 2013, pp. 854–860. URL: http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6997.

[12] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Multi-shot asp solving with clingo, Theory and Practice of Logic Programming 19 (2019) 27–82. doi:10.1017/S1471068418000054.

[13] T. Janhunen, I. Niemelä, M. Sevalnev, Computing stable models via reductions to difference logic, in: E. Erdem, F. Lin, T. Schaub (Eds.), Proc. of LPNMR 2009, LNCS, Springer, 2009, pp. 142–154. URL: https://doi.org/10.1007/978-3-642-04238-6_14. doi:10.1007/978-3-642-04238-6_14.

[14] T. Janhunen, Representing normal programs with clauses, in: Proceedings of the 16th European Conference on Artificial Intelligence, ECAI'04, IOS Press, NLD, 2004, p. 358–362.

[15] M. Gebser, T. Janhunen, J. Rintanen, SAT modulo graphs: Acyclicity, in: E. Fermé, J. Leite (Eds.), Proc. of JELIA 2014, LNCS, Springer, 2014, pp. 137–151. URL: https://doi.org/10.1007/978-3-319-11558-0_10. doi:10.1007/978-3-319-11558-0_10.

[16] V. Lifschitz, A. Razborov, Why are there so many loop formulas?, ACM Trans. Comput. Logic 7 (2006) 261–268. URL: https://doi.org/10.1145/1131313.1131316. doi:10.1145/1131313.1131316.

[17] J. Ellson, E. Gansner, L. Koutsofios, S. North, G. Woodhull, Graphviz - open source graph drawing tools, in: Graph Drawing 2001, Springer-Verlag, Berlin, 2002, pp. 483–484. doi:10.1007/3-540-45848-4_57.

[18] R. Harrop, On disjunctions and existential statements in intuitionistic systems of logic., Mathematische Annalen 132 (1956/57) 347–361. URL: http://eudml.org/doc/160525.