`STeLP` – A Tool for Temporal Answer Set Programming[⋆]

Pedro Cabalar and Martín Diéguez

Department of Computer Science,
University of Corunna (Spain)
{`cabalar,martin.dieguez`}`@udc.es`

Abstract. In this paper we present STeLP, a solver for Answer Set Programming with temporal operators. Taking as an input a particular kind of logic program with modal operators (called Splitable Temporal Logic Program), STeLP obtains its set of temporal equilibrium models (a generalisation of stable models for this extended syntax). The obtained set of models is represented in terms of a deterministic Büchi automaton capturing the complete program behaviour. In small examples, this automaton can be graphically displayed in a direct and readable way. The input language provides a set of constructs which allow a simple definition of temporal logic programs, including a special syntax for action domains that can be exploited to simplify the graphical output. STeLP combines the use of a standard ASP solver with a linear temporal logic model checker in order to find all models of the input theory.

1 Introduction

The use of Answer Set Programming (ASP) tools to represent temporal scenarios for Non-Monotonic Reasoning (NMR) has some important limitations. ASP solvers are commonly focused on finite domains and so, representation of time usually involves a finite bound. Typically, variables ranging over transitions or time instants are grounded for a sequence of numbers $0, 1, \dots, n$ where n is a finite length we must fix beforehand. As a result, for instance, we cannot check the non-existence of a plan for a given planning problem, or that some transition system satisfies a given property for any of its possible executions, or that two representations of the same dynamic scenario are *strongly equivalent* (that is, they always have the same behaviour for any considered narrative length) to mention three relevant examples.

To overcome these limitations, [1] introduced an extension of ASP called *Temporal Equilibrium Logic*. This formalism combines Equilibrium Logic [2] (a logical characterisation of ASP) with Linear Temporal Logic (LTL) [3] and provides a definition of the *temporal equilibrium models* (analogous to stable models) for any arbitrary temporal theory.

⋆ This research was partially supported by Spanish MEC project TIN2009-14562-C05-04 and Xunta de Galicia project INCITE08-PXIB105159PR.

In this paper we introduce STeLP[1], a system for computing the temporal equilibrium models of a particular class of temporal theories called *Splitable Temporal Logic Programs* (STLP). This class suffices to cover most frequent examples in ASP for dynamic domains. More importantly, the temporal equilibrium models of an STLP have been shown to be computable in terms of a regular LTL theory (see the companion paper [4]). This feature is exploited by STeLP to call an LTL model checker as a backend.

2 Splitable Temporal Logic Programs

Temporal Equilibrium Logic (TEL) shares the syntax of propositional LTL, that is, propositional formulas plus the unary temporal operators[2] $\Box$ (read as "always"), $\Diamond$ ("eventually") and $\bigcirc$ ("next"). TEL is defined in two steps: first, we define the monotonic logic of *Temporal Here-and-There* (THT); and second, we select TEL models as some kind of minimal THT-models obtaining non-monotonicity. For further details, see [5].

Definition 1 (STLP). *An* initial rule *is an expression of one of the forms:*

$$A_1 \wedge \cdots \wedge A_n \wedge \neg A_{n+1} \wedge \cdots \wedge \neg A_m \rightarrow A_{m+1} \vee \cdots \vee A_s \tag{1}$$

$$B_1 \wedge \cdots \wedge B_n \wedge \neg B_{n+1} \wedge \cdots \wedge \neg B_m \rightarrow \bigcirc A_{m+1} \vee \cdots \vee \bigcirc A_s \tag{2}$$

where A_i are atoms and each B_j can be an atom p or the formula $\bigcirc p$. A dynamic rule *has the form $\Box r$ where r is an initial rule. A* Splitable Temporal Logic Program *(STLP) Π is a set of (initial and dynamic) rules.* ⊠

The *body* (resp. *head*) of a rule is the antecedent (resp. consequent) of its implication connective. As usual, a rule with empty body (that is, $\top$) is called a *fact*, whereas a rule with empty head (that is, $\bot$) is called a called a *constraint*. An initial fact $\top \rightarrow \alpha$ is just written as α. The following theory Π_1 is an STLP:

$$\neg a \wedge \bigcirc b \rightarrow \bigcirc a \qquad \Box(a \rightarrow b) \qquad \Box(\neg b \rightarrow \bigcirc a)$$

In [4] it is shown how the temporal equilibrium models of an STLP Π correspond to the LTL models of $\Pi \cup LF(\Pi)$ where $LF(\Pi)$ are loop formulas adapted from the result in [6]. For further details and a precise definition, see [4].

3 The Input Language

The input programs of STeLP adopt the standard ASP notation for conjunction, negation and implication, so that, an initial rule like (1) is represented as:

$$A_{m+1}\ \texttt{v}\ldots\texttt{v}\ A_s\ \texttt{:-}\ A_1\texttt{,}\ldots\texttt{,}A_n\texttt{,}\ \texttt{not}\ A_{n+1}\texttt{,}\ldots\texttt{,}\ \texttt{not}\ A_m$$

Operator '$\bigcirc$' is represented as 'o' whereas a dynamic rule like $\Box(\alpha \rightarrow \beta)$ is written as β `::-` α. Using this notation, program Π_1 becomes:

[1] A STeLP web version is available at `http://kr.irlab.org/stelp`

[2] As shown in [5], the LTL binary operators $\mathcal{U}$ ("until") and $\mathcal{R}$ ("release") can be removed by introducing auxiliary atoms.

```
o a :- not a, o b.        b ::- a.        o a ::- not b.
```

Constraints in STeLP are more general than in STLP: their body can include any arbitrary combination of propositional connectives with o, always (standing for $\Box$) and until (standing for $\mathcal{U}$). The empty head $\bot$ is not represented. For instance, $\Box(\bigcirc a \wedge \neg b \rightarrow \bot)$ and $(\Box\neg g) \rightarrow \bot$ are constraints written as:

```
::- o a, not b.        :- always not g.
```

In STeLP we can also use rules where atoms have variable arguments like p(X_1,...,X_n) and, as happens with most ASP solvers, these are understood as abbreviations of all their ground instances. A kind of *safety* condition is defined for variables occurring in a rule. We will previously distinguish a family of predicates, called *static*, that satisfy the property $\Box(\ p(\overline{X}) \leftrightarrow \bigcirc p(\overline{X})\ )$ for any tuple of elements $\overline{X}$. These predicates are declared using a list of pairs *name/arity* preceded by the keyword static. All built-in relational operators =, !=, <, >, <=, >= are implicitly defined as static, having their usual meaning. An initial or dynamic rule is *safe* when:

1. Any variable X occurring in a rule $B \rightarrow H$ or $\Box(B \rightarrow H)$ occurs in some positive literal in B for some static predicate p.
2. Initial rules of the form $B \rightarrow H$ where at least one static predicate occurs in the head H only contain static predicates (these are called *static rules*).

Since static predicates must occur in any rule, STeLP allows defining global variable names with a fixed domain, in a similar way to the lparse[3] directive #domain. For instance, the declaration domain switch(X). means that any rule referring to variable X is implicitly extended by including an atom switch(X) in its body. All predicates used in a domain declaration must be static – as a result, they will be implicitly declared as static, if not done elsewhere.

As an example, consider the classical puzzle where we have a wolf w, a sheep s and a cabbage c at one bank of a river. We have to cross the river carrying at most one object at a time. The wolf eats the sheep, and the sheep eats the cabbage, if no people around. Action m(X) means that we move some item w,s,c from one bank to the other. We assume that the boat is always switching its bank from one state to the other, so when no action is executed, this means we moved the boat without carrying anything. We will use a unique fluent at(Y,B) meaning that Y is at bank B being Y an item or the boat b. The complete encoding is shown in Figure 1.

A feature that causes a difficult reading of the obtained automaton for a given STLP is that all the information is represented by formulas that occur as transition labels, whereas states are just given a meaningless name. As opposed to this, in an actions scenario, one would expect that states displayed the fluents information and transitions only contained the actions execution. To make the automaton closer to this more natural representation, we can distinguish predicates representing actions and fluents. For instance, in the previous example,

[3] http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz

```
% Domain predicates
domain item(X), object(Y).
static opp/2.     fluent at/2.     action m/1.
opp(l,r). opp(r,l).     item(w). item(s). item(c).
object(Z) :- item(Z).   object(b).
o at(X,A) ::- at(X,B), m(X), opp(A,B).       % Effect axiom for moving
o at(b,A) ::- at(b,B), opp(A,B).             % The boat is always moving
::- m(X), at(b,A), at(X,B), opp(A,B).        % Action executability
::- at(Y,A), at(Y,B), opp(A,B).              % Unique value constraint
o at(Y,A) ::- at(Y,A), not o at(Y,B),opp(A,B).% Inertia
::- at(w,A), at(s,A), at(b,B), opp(A,B).     % Wolf eats sheep
::- at(s,A), at(c,A), at(b,B), opp(A,B).     % Sheep eats cabbage
a(X) ::- not m(X).                           % Choice rules for action
m(X) ::- not a(X).                           %    execution
::- m(X), item(Z), m(Z), X != Z.             % Non-concurrent actions
at(Y,l).                                     % Initial state
g ::- at(w,r), at(s,r), at(c,r).             % Goal predicate
:- always not g.                             % Goal must be satisfied
```

Fig. 1. Wolf-sheep-cabbage puzzle in STeLP

we would further declare: `action m/1. fluent at/2.` STeLP uses this information so that when all the outgoing transitions from a given state share the same information for fluents, this is information is shown altogether inside the state, and removed from the arc labels. Besides, any symbol that is not an action or a fluent is not displayed (they are considered as auxiliary). As a result of these simplifications, we may obtain several transitions with the same label: if so, they are collapsed into a single one.

4 Implementation

STeLP is a Prolog application that interacts with the standard ASP solver DLV[4] and the LTL model checker SPOT[5]. As shown in Figure 2, it is structured in several modules we describe next.

In a first step, STeLP parses the input program, detecting static and domain predicates, checking safety of all rules, and separating the static rules. It also appends static predicates to rule bodies for all variables with global domain. The set of static rules is fed to DLV to generate some model (among possible) that will provide the extension for all static predicates. The number of static models that STeLP will consider is given as a command line argument. Each static model will generate a different ground program and a different automaton. Once a static model is fixed, STeLP grounds the non-static rules input program. Each ground instance is evaluated and, if the body of the ground rule becomes false, the rule is deleted. Otherwise all static predicates of the rule are deleted from its body.

[4] http://www.dbai.tuwien.ac.at/proj/dlv/

[5] http://spot.lip6.fr/

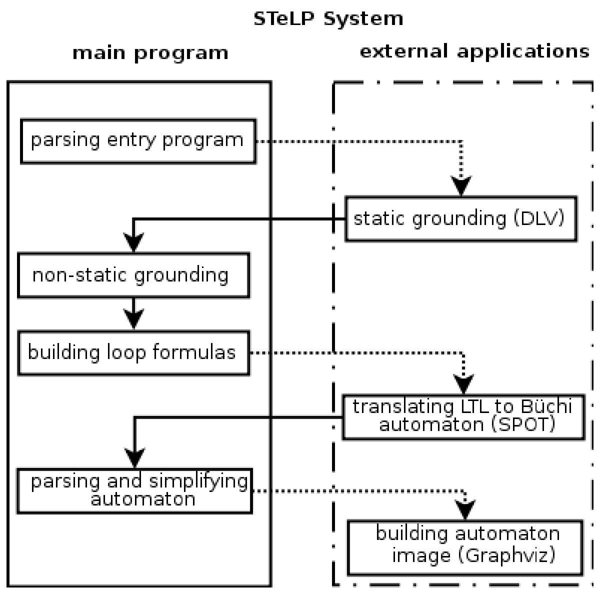

Fig. 2. Structure of `STeLP` system

The next step computes the loop formulas for the ground STLP we have obtained by constructing a dependency graph and obtaining its strongly connected components (the loops) using Tarjan's algorithm [7]. The STLP plus its loop formulas are then used as input for the LTL solver `SPOT`, which returns a deterministic Büchi automaton. Finally, `STeLP` parses and, if actions and fluents are defined, simplifies the automaton as described before. The tool `Graphviz`[6] is used for generating a graphical representation. For instance, our wolf-sheep-cabbage example throws the diagram in Figure 3. As an example of non-existence of plan, if we further include the rule `::- at(w,r), at(c,r), at(b,l)` meaning that we cannot leave the wolf and the cabbage alone in the right bank, then the problem becomes unsolvable (we get a Büchi automaton with no accepting path).

`STeLP` is a first prototype without efficiency optimisations – exhaustive benchmarking is left for future work. Still, to have an informal idea of its current performance, the example above was solved in 0.208 seconds[7]. The automaton for the whole behaviour of the classical scenario involving 3 missionaries and 3 cannibals is solved in 160.358 seconds. `STeLP` is able to show that this same scenario has no solution for 4 individuals in each group, but the answer is obtained in more than 1 hour.

[6] `http://www.graphviz.org/`

[7] Using an Intel Xeon 2.4 GHz, 16 GB RAM and 12 MB of cache size, with software tools SPOT 0.7.1, DLV oct-11-2007 and SWI Prolog 5.8.0.

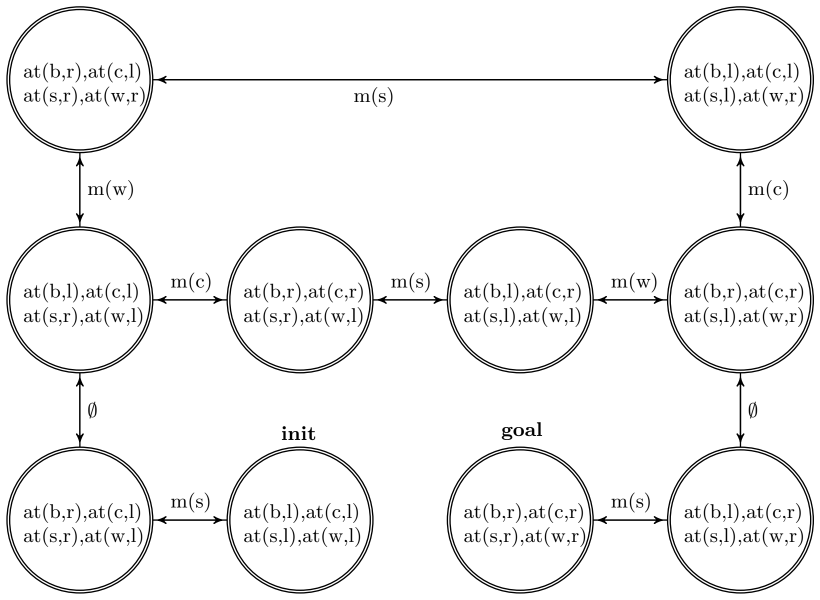

Fig. 3. Automaton for the wolf-sheep-cabbage example

References

1. Cabalar, P., Vega, G.P.: Temporal equilibrium logic: a first approach. In: Moreno Díaz, R., Pichler, F., Quesada Arencibia, A. (eds.) EUROCAST 2007. LNCS, vol. 4739, pp. 241–248. Springer, Heidelberg (2007)
2. Pearce, D.: A new logical characterisation of stable models and answer sets. In: Dix, J., Przymusinski, T.C., Moniz Pereira, L. (eds.) NMELP 1996. LNCS, vol. 1216. Springer, Heidelberg (1997)
3. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer, Heidelberg (1991)
4. Aguado, F., Cabalar, P., Pérez, G., Vidal, C.: Loop formulas for splitable temporal logic programs. In: Delgrande, J., Faber, W. (eds.) LPNMR 2011. LNCS (LNAI), vol. 6645, pp. 78–90. Springer, Heidelberg (2011), `http://www.dc.fi.udc.es/~cabalar/lfstlp.pdf`
5. Cabalar, P.: A normal form for linear temporal equilibrium logic. In: Janhunen, T., Niemelä, I. (eds.) JELIA 2010. LNCS, vol. 6341, pp. 64–76. Springer, Heidelberg (2010)
6. Ferraris, P., Lee, J., Lifschitz, V.: A generalization of the Lin-Zhao theorem. Annals of Mathematics and Artificial Intelligence 47, 79–101 (2006)
7. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM Journal on Computing 1(2), 146–160 (1972)