Verification of action theories in ASP: a complete Bounded Model Checking approach

Laura Giordano[1], Alberto Martelli[2], and Daniele Theseider Dupré[1]

[1] DISIT, Università del Piemonte Orientale {laura.giordano,dtd}@mfn.unipmn.it
[2] Dipartimento di Informatica, Università di Torino mrt@di.unito.it

Abstract. Temporal logics are well suited for reasoning about actions, as they allow for the specification of domain descriptions including temporal constraints as well as for the verification of temporal properties of the domain. This paper focuses on the verification of action theories formulated in a temporal extension of *answer set programming* which combines ASP with a dynamic linear time temporal logic. The paper proposes an approach to bounded model checking (BMC) which exploits the Büchi automaton construction while searching for a counterexample, with the aim of achieving completeness. The paper provides an encoding in ASP of the temporal action domains and of BMC of DLTL formulas.

1 Introduction

Temporal logics have been extensively used in the specification and verification of action domains in many fields, from planning to web services. In planning, both CTL [22, 25] and LTL [4, 3] have been used in the specification of temporally extended goals. The need for state trajectory constraints has been advocated in PDDL3 [15]. [2] exploits a first order linear temporal logic for defining domain dependent search control knowledge in the planner TLPlan, and in [11] strong fairness constraints expressed in LTL are used to restrict nondeterminism in generalized planning. LTL has been used in the verification of agent interaction protocols [18] and for enforcing regulations in automated Web service composition [26]. In the context of reasoning about action, [9] has introduced a second order extension of the temporal logic CTL*, $\mathcal{ESG}$, to reason about non-terminating Golog programs.

In this paper, we start from the temporal action theories introduced in [19], formulated in a temporal extension of *answer set programming* (ASP [14]), and we exploit Bounded Model Checking (BMC) techniques for the verification of properties of such action theories. BMC [5] is an efficient model checking techniques which does not require a tableau or automaton construction. Given a system model (a transition system) and a property to be checked, it searches for a counterexample of the property as a path of length k, generating a propositional formula that is satisfiable iff such a counterexample exists. The bound k on the length of the path is iteratively increased and, if no counterexample exists, the procedure never stops. As a consequence, bounded model checking provides only a partial decision procedure for checking validity. Techniques for achieving completeness have been described in [5], where upper bounds for k are determined for some classes of properties, namely unnested properties. To deal with completeness, [8] proposes a *semantic* translation scheme, based on Büchi automata.

In [23] Helianko and Niemelä developed a compact encoding of bounded model checking of LTL formulas as the problem of finding stable models of logic programs. Since ASP naturally accommodates for reasoning about actions, in [19] this encoding is extended to deal with Dynamic Linear Time Temporal Logic (DLTL) formulas, for reasoning about action theories including complex actions and programs. These papers do not address the problem of achieving completeness.

In this paper we propose an alternative encoding of BMC of DLTL formulas in ASP, with the aim of achieving completeness. Unlike [23, 19], here the search for a counterexample exploits the Büchi automaton construction [16] as well as the transition system. Unlike [8], a "counterexample" path is searched for, without assuming that the Büchi automaton is constructed in advance. Our counterexample is an accepting path of the product Büchi automaton which can be finitely represented as a *(k,l)-loop* , i.e., a finite path of length k terminating in a loop back to a previous state l, in which the states are all distinct from each other. The procedure for verifying a given property searches for a *(k,l)-loop*, providing a counterexample to the property, increasing k until either a counterexample is found, or no *(k,l)-loop* of length greater or equal to k can be found.

As in [19], verification is performed on a transition system provided by a domain description in a temporal action theory, and our BMC approach is used for proving properties of domain descriptions. The action theory is given in a temporal extension of ASP, based on the generalization of the notion of *answer set* [14] to *temporal answer sets*. The temporal properties of a domain description can be proved by combining the construction of temporal extensions of the domain with the verification of their properties, according to a tableaux-based procedure which provides an encoding of BMC in ASP. The proposed approach provides a decision procedure for the verification of satisfiability and validity properties of an action domain in a temporal action theory.

The outline of the paper is the following. First, we introduce the temporal action language and its answer sets, and we introduce verification problems for action theories. We then describe our approach to action theory verification by BMC. Finally we provide an ASP encoding of BMC and discuss related work.

2 Temporal Action Theories

In this paper we refer to a formulation of DLTL (dynamic linear time temporal logic), in [24], where the next state modality is indexed by actions and the until operator $\mathcal{U}^{\pi}$ is indexed by a program π which, as in PDL, can be any regular expression built from atomic actions using sequence (;), nondeterministic choice $(+)$ and finite iteration $(*)$.

Let $\Sigma = \{a_1, \ldots, a_n\}$ be a finite non-empty alphabet of actions. From the until operator, the derived modalities $\langle\pi\rangle$, $[a]$, $\bigcirc$ (next), $\mathcal{U}$, $\Diamond$ and $\Box$ can be defined as follows: $\langle\pi\rangle\alpha \equiv \top\mathcal{U}^{\pi}\alpha$, $[a]\alpha \equiv \neg\langle a\rangle\neg\alpha$, $\bigcirc\alpha \equiv \bigvee_{a\in\Sigma}\langle a\rangle\alpha$, $\alpha\mathcal{U}\beta \equiv \alpha\mathcal{U}^{\Sigma^*}\beta$, $\Diamond\alpha \equiv \top\mathcal{U}\alpha$, $\Box\alpha \equiv \neg\Diamond\neg\alpha$, where α is a formula and, in $\mathcal{U}^{\Sigma^*}$, Σ is taken to be a shorthand for the program $a_1 + \ldots + a_n$

2.1 Temporal Action Language

Let $\mathcal{L}$ be a first order language which includes a finite number of constants and variables, but no function symbol. Let $\mathcal{P}$ be the set of predicate symbols, Var the set of variables and C the set of constant symbols. We call *fluents* atomic literals of the form $p(t_1, \ldots, t_n)$, where, for each i, $t_i \in Var \cup C$. A *simple fluent literal* (or *s-literal*) l is an atomic literals $p(t_1, \ldots, t_n)$ or its negation $\neg p(t_1, \ldots, t_n)$. We denote by Lit_S the set of all simple fluent literals. Lit_T is the set of *temporal fluent literals*: if $l \in Lit_S$, then $[a]l, \bigcirc l \in Lit_T$, where a is an action name (an atomic proposition, possibly containing variables), and $[a]$ and $\bigcirc$ are the temporal operators introduced in the previous section. Let $Lit = Lit_S \cup Lit_T \cup \{\perp\}$, where $\perp$ represents the inconsistency. Given a (simple or temporal) fluent literal l, $not\ l$ represents the default negation of l. A (simple or temporal) fluent literal possibly preceded by a default negation, will be called an *extended fluent literal*.

A *domain description* Π is a set of laws describing the effects of actions and their executability preconditions. The laws are formulated as rules of a temporally extended logic programming language. Rules have the form

$$l_0 \leftarrow l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n \qquad (1)$$

where the l_i's are either simple fluent literals or temporal fluent literals, with the following constraints: (i) If l_0 is a simple literal, then the body cannot contain temporal literals; (ii) If $l_0 = [a]l$, then the temporal literals in the body must have the form $[a]l'$; (iii) If $l_0 = \bigcirc l$, then the temporal literals in the body must have the form $\bigcirc l'$. As usual in ASP, the rules with variables will be used as a shorthand for the set of their ground instances.

A *state*, informally, is a set of ground fluent literals closed with respect to the rules above (see Section 2.2). A state is said to be *consistent* if it is not the case that both f and $\neg f$ belong to the state, or that $\perp$ belongs to the state. A state is said to be *complete* if, for each fluent name $p \in \mathcal{P}$, either p or $\neg p$ belong to the state. The execution of an action in a state may possibly change the values of fluents in the state through its direct and indirect effects, thus giving rise to a new state. We assume that a law as (1) can be applied in all states, while when prefixed with the **Init**, only applies to the initial state.

Example 1. This example describes a mail delivery agent, which checks if there is mail in the mailbox of employees and delivers mail to them. The actions in Σ are: $sense$ (the agent verifies if there is mail in any of the mailboxes), $deliver(E)$ (the agent delivers the mail to employee E), $wait$. The fluent names are $mail(E)$ (there is mail in the mailbox of E). Π contains the following immediate effects and persistency laws:

$[deliver(E)]\neg mail(E)$
$[sense]mail(E) \leftarrow\ not\ [sense]\neg mail(E)$
$\bigcirc mail(E) \leftarrow mail(E), not \bigcirc \neg mail(E)$
$\bigcirc \neg mail(E) \leftarrow \neg mail(E), not \bigcirc mail(E)$

Their meaning is (in the order) that: after delivering the mail to E, there is no mail for E any more; the action $sense$ may (non-monotonically) cause $mail(E)$ to become true. The last two rules define the persistency of fluent $mail$.

Observe that the persistency laws interact with the immediate effect laws above. The execution of $sense$ in a state in which there is no mail for some E ($\neg mail(E)$), may either lead to a state in which $mail(E)$ holds (by the second action law) or to a state in which $\neg mail(E)$ holds (by persistency of $\neg mail(E)$). Thus, $sense$ is a nondeterministic action. The following precondition laws:

$[deliver(E)]\ \bot \leftarrow \neg mail(E)$
$[wait]\ \bot \leftarrow mail(E)$
specify that, if there is no mail for E, $deliver(E)$ is not executable, while, if there is mail for E, $wait$ is not executable.

We assume that there are only two employees, a and b and, in the initial state, there is mail for a and not for b, i.e. Π includes $\mathbf{Init}\ mail(a)$ and $\mathbf{Init}\ \neg mail(b)$.

The language is also well suited to describe causal dependencies among fluents [19] by the definition of *static* and *dynamic causal laws* similar to the ones in the action languages $\mathcal{K}$ [12] and $\mathcal{C}^+$ [20].

2.2 Temporal Answer Sets

In this section, we recall the notion of *temporal answer set* in [19], which extends the notion of *answer set* [14], and we state a new result on the transition system associated with a domain description. To this purpose, we let Π be the ground instantiation of the domain description, and Σ the set of all the ground instances of the action names in Π.

A temporal interpretation is defined as a pair (σ, S), where $\sigma \in \Sigma^\omega$ is a sequence of actions and S is a consistent set of ground literals of the form $[a_1; \ldots; a_k]l$, where $a_1 \ldots a_k$ is a prefix of σ and l is a ground simple fluent literal, meaning that l holds in the state obtained by executing $a_1 \ldots a_k$. S is *consistent* iff it is not the case that both $[a_1; \ldots; a_k]l \in S$ and $[a_1; \ldots; a_k]\neg l \in S$, for some l, or $[a_1; \ldots; a_k]\bot \in S$. A temporal interpretation (σ, S) is said to be *total* if either $[a_1; \ldots; a_k]p \in S$ or $[a_1; \ldots; a_k]\neg p \in S$, for each $a_1 \ldots a_k$ prefix of σ and for each fluent name p.

The notion of satisfiability of a rule in a temporal interpretation (σ, S), as well as the notion of *reduct $\Pi^{(\sigma,S)}$ of (a domain description) Π relative to* (σ, S) can be defined as natural extensions of Gelfond and Lifschitz' ones [14]. With these notions, a temporal answer set of Π is defined as a temporal interpretation (σ, S) such that S is minimal (in the sense of set inclusion) among the S' such that (σ, S') is a partial interpretation satisfying the rules in the reduct $\Pi^{(\sigma,S)}$.

The case of total temporal answer sets is of special interest, as a total temporal answer set (σ, S) can be regarded as temporal model (σ, V_S), where, for each finite prefix $a_1 \ldots a_k$ of σ, $V_S(a_1, \ldots, a_k) = \{p : [a_1, \ldots, a_k]p \in S\}$. In the following, we restrict our consideration to domain descriptions Π, such that all the answer sets of Π are total. If the initial state is not complete, we consider all the possible ways to complete the initial state by introducing in Π, for each fluent name f, the rules: $\mathbf{Init}\ f \leftarrow not\ \neg f$ and $\mathbf{Init}\ \neg f \leftarrow not\ f$.

A total temporal interpretation (σ, S) provides, for each prefix $a_1 \ldots a_k$, a complete state corresponding to that prefix. We denote by $w^{(\sigma,S)}_{a_1 \ldots a_k}$ the state obtained by the execution of the actions $a_1 \ldots a_k$ in the sequence, namely $w^{(\sigma,S)}_{a_1 \ldots a_k} = \{l : [a_1; \ldots; a_k]l \in S\}$.

Given a domain description Π over Σ with total answer sets, a *transition system* (W, I, T) can be associated with Π as follows: (i) W is the set of all the possible consistent and complete states of the domain description; (ii) I is the set of all the states in W satisfying the initial state laws in Π; (iii) $T \subseteq W \times \Sigma \times W$ is the set of all triples (w, a, w') such that: $w, w' \in W$, $a \in \Sigma$ and for some total answer set (σ, S) of Π: $w = w^{(\sigma,S)}_{[a_1;\ldots;a_h]}$ and $w' = w^{(\sigma,S)}_{[a_1;\ldots;a_h;a]}$, for some h.

It is possible to show that the next states of a given state w in the transition system (W, I, T) above only depend on the state w. Let Π_w be the domain description obtained form Π by removing all the laws prefixed by **Init** while adding to Π **Init** l, for all $l \in w$.

Proposition 1. *Let w be a state in W which is reachable form an initial state by the action sequence $a_1 \ldots a_h$. If $(w, a, w') \in T$, then there is an answer set (σ', S') of Π_w, such that (1) $\sigma = a_1 \ldots a_h \sigma'$ and (2) $[a]l \in S'$ iff $l \in w'$. Vice versa, if there is an answer set (σ', S') of Π_w satisfying conditions (1) and (2) above, then $(w, a, w') \in T$.*

Proposition 1 guarantees that, given a state w and an action a, a next state function $nextTSstate$ can be defined to compute all the states reachable in the transition system from w by a. Such a function is used in the following to describe the BMC construction.

2.3 Verification of Enriched Domain Descriptions

As a total temporal answer set of a domain description can be interpreted as an DLTL model, it is easy to combine domain descriptions with DLTL formulas. This can be done by adding to the domain description Π a set of DLTL formulas $\mathcal{C}$ used as constraints on the executions of the domain description. We denote by $(\Pi, \mathcal{C})$ the enriched domain description, and we define the *extensions of* $(\Pi, \mathcal{C})$ to be the temporal answer sets (σ, S) of Π satisfying the constraints $\mathcal{C}$. For example,

$\langle begin \rangle \top$
$\Box [begin] \langle sense; (deliv(a) + deliv(b) + wait); begin \rangle \top$

impose that the agent continuously executes a loop where it senses mail and delivers the mail. DLTL formulas can be used to encode properties to be verified on the enriched domain description. We may want to check that, if there is mail for a, the agent will eventually deliver it, i.e.: $\Box(mail(a) \supset \Diamond \neg mail(a))$. This does not hold, as there is a possible scenario in which there is always mail for a and for b, but the mail is repeatedly delivered to b and never to a.

Given an enriched domain description $(\Pi, \mathcal{C})$, some problems, e.g. planning, can be formulated as *satisfiability* of a formula φ, and others, such as the one in the example above, as validity of a formula φ. Usually, the validity of a property φ formulated as a DLTL formula is reduced to the *unsatisfiability* of $\neg\varphi$. In this case, if a model satisfying $\neg\varphi$ is found, it represents a counterexample to the validity of φ.

3 Model Checking

Satisfiability and validity problems can be solved by means of *model checking* techniques. The standard approach to model checking for LTL is based on Büchi automata. The satisfiability problem for a LTL formula α can be solved by constructing a Büchi automaton $\mathcal{B}_\alpha$ [16] such that the language of ω-words accepted by $\mathcal{B}_\alpha$ is non-empty if and only if α is satisfiable.

Given a system modeled by a transition system TS, which corresponds to a Büchi automaton $\mathcal{B}_{TS}$, *model checking* verifies that the property α holds for the system, by constructing the *product automaton* of $\mathcal{B}_{TS}$ and $\mathcal{B}_{\neg\alpha}$, and by checking for emptiness of the accepted language.

Biere et al. [5] showed that model checking can be more efficient if, instead of building the product automaton, a path of the transition system satisfying $\neg\alpha$ is searched for. This technique is called *bounded model checking* (BMC), since it looks for infinite paths which can be represented as a finite path of length k with a back loop from state k to a previous state l in the path (a *(k,l)-loop*); the search proceeds iteratively, increasing the length k until a model satisfying α is found — if one exists.

A BMC problem can be efficiently reduced to a propositional satisfiability problem or to an ASP problem [23]. If no model exists and the transition system contains a loop, the iterative procedure in general does not stop, i.e., it is a partial decision procedure for validity. Techniques for achieving completeness are described in [5] for some kinds of LTL formulas.

4 Bounded Model Checking with Büchi Automata

In this paper, we propose an approach to model checking which combines the advantages of BMC, in particular the possibility of formulating it easily and efficiently as an ASP problem, with the advantages of reasoning on the product Büchi automaton described above, mainly its completeness.

In the following we show how to adapt the procedure for building a Büchi automaton corresponding to a given DLTL formula [17] to the "on-the-fly" construction of the *product* Büchi automaton, and we show how this construction can be used to build a *(k,l)-loop* corresponding to a run of the product Büchi automaton.

In the following construction we assume that, as in [17], $until$ formulas are indexed with finite automata rather than regular expressions. Thus, we have $\alpha\mathcal{U}^{\mathcal{A}(q)}\beta$ instead of $\alpha\mathcal{U}^{\pi}\beta$, where $\mathcal{L}(\mathcal{A}(q)) = [[\pi]]$. We denote with $\mathcal{A}(q)$ a finite automaton $\mathcal{A}$ with initial state q. The following equivalences hold for the until operator [24]:

$$\alpha\mathcal{U}^{\mathcal{A}(q)}\beta \equiv (\beta \vee (\alpha \wedge \bigvee_{a\in\Sigma}\langle a\rangle \bigvee_{q'\in\delta(q,a)} \alpha\mathcal{U}^{\mathcal{A}(q')}\beta)) \quad (q \text{ is a final state of } \mathcal{A})$$
$$\alpha\mathcal{U}^{\mathcal{A}(q)}\beta \equiv (\alpha \wedge \bigvee_{a\in\Sigma}\langle a\rangle \bigvee_{q'\in\delta(q,a)} \alpha\mathcal{U}^{\mathcal{A}(q')}\beta) \quad (q \text{ is not a final state of } \mathcal{A})$$

The construction of the nodes makes use of tableau rules which handle DLTL *signed formulas*, i.e. formulas prefixed with the symbol **T** or **F**. These rules are applied to a set of formulas[1] as follows:

– $\phi \Rightarrow \psi_1, \psi_2$, if ϕ belongs to the set of formulas, then add ψ_1 and ψ_2 to the set
– $\phi \Rightarrow \psi_1|\psi_2$, if ϕ belongs to the set of formulas, then make two copies of the set and add ψ_1 to one of them and ψ_2 to the other one.

The rules are the following:

Tor:	$\mathbf{T}(\alpha \vee \beta) \Rightarrow \mathbf{T}\alpha|\mathbf{T}\beta$
For:	$\mathbf{F}(\alpha \vee \beta) \Rightarrow \mathbf{F}\alpha, \mathbf{F}\beta$
Tneg:	$\mathbf{T}\neg\alpha \Rightarrow \mathbf{F}\alpha$
Fneg:	$\mathbf{F}\neg\alpha \Rightarrow \mathbf{T}\alpha$
TuntilFS:	$\mathbf{T}\alpha\mathcal{U}^{\mathcal{A}(q)}\beta \Rightarrow \mathbf{T}(\beta \vee (\alpha \wedge \bigvee_{a\in\Sigma}\langle a\rangle \bigvee_{q'\in\delta(q,a)} \alpha\mathcal{U}^{\mathcal{A}(q')}\beta))$ (q final state)

[1] In this section "formula" means "signed DLTL formula".

TuntilNFS: $\mathbf{T}\alpha\mathcal{U}^{\mathcal{A}(q)}\beta \Rightarrow \mathbf{T}(\alpha \wedge \bigvee_{a\in\Sigma}\langle a\rangle\bigvee_{q'\in\delta(q,a)} \alpha\mathcal{U}^{\mathcal{A}(q')}\beta)$ (q not final state)
FuntilFS: $\mathbf{F}\alpha\mathcal{U}^{\mathcal{A}(q)}\beta \Rightarrow \mathbf{F}(\beta \vee (\alpha \wedge \bigvee_{a\in\Sigma}\langle a\rangle \bigvee_{q'\in\delta(q,a)} \alpha\mathcal{U}^{\mathcal{A}(q')}\beta))$ (q final state)
FuntilNFS: $\mathbf{F}\alpha\mathcal{U}^{\mathcal{A}(q)}\beta \Rightarrow \mathbf{F}(\alpha \wedge \bigvee_{a\in\Sigma}\langle a\rangle\bigvee_{q'\in\delta(q,a)} \alpha\mathcal{U}^{\mathcal{A}(q')}\beta)$ (q not final state)

We use a function *tableau* which takes as input a set of formulas s, adds to it $\mathbf{T}\bigvee_{a\in\Sigma}\langle a\rangle\top$, and returns a (possibly empty) set of sets of formulas, obtained by repeatedly applying the above rules (by possibly creating new sets) until all non-elementary formulas in all sets have been expanded. We call *elementary formulas* the formulas of the form $\mathbf{T}\phi$ or $\mathbf{F}\phi$ where ϕ is either $\top$, or $\bot$, or a proposition or $\langle a\rangle\alpha$. Formula $\mathbf{T}\bigvee_{a\in\Sigma}\langle a\rangle\top$ makes explicit that in DLTL each state must be followed by a next state.

If the expansion of a set of formulas produces an inconsistent set, then this set is deleted. A set of formulas s is *inconsistent* in the following cases: (i) $\mathbf{T}\bot \in s$; (ii) $\mathbf{F}\top \in s$; (iii) $\mathbf{T}\alpha \in s$ and $\mathbf{F}\alpha \in s$; (iv) $\mathbf{T}\langle a\rangle\alpha \in s$ and $\mathbf{T}\langle b\rangle\beta \in s$ with $a \neq b$, because in a linear time logic two different actions cannot be executed in the same state.

We describe now how to build a path of the product automaton, which is constructed by the BMC procedure while searching for a counterexample. Each state s of the path is a tuple $s = (\mathcal{F}, w, x, f)$, where $\mathcal{F}$ is an expanded set of formulas, w is a state of the transition system whose literals are represented as signed formulas, $x \in \{0,1\}$ and $f \in \{\downarrow, \checkmark\}$ are used to track fulfillment of until formulas, as we will describe below.

Given a domain description Π with the associated transition system TS, and a DLTL formula α describing constraints and properties to be proved, the initial states will have the form $(\mathcal{F}_0, w_0, 0, \checkmark)$, where $\mathcal{F}_0$ is a set of formulas obtained by applying function $tableau$ to α, and w_0 is an initial state of TS, such that $\mathcal{F}_0 \cup w_0$ is consistent.

Transitions of the product automaton are defined by function $next\_states(s,a)$, defined in Figure 2, which returns the set of successor states of s after a. This function makes use of the functions $nextTSstates(w,a)$, which returns the set of the states of the transition system TS reached with a transition a from state w, and $next\mathcal{F}(\mathcal{F},a)$, which returns a set of formulas obtained by propagating the formulas in $\mathcal{F}$ through action a. Function $next\mathcal{F}$ is defined in Figure 1. This function first checks whether it is possible to execute action a from $\mathcal{F}$, then propagates elementary temporal formulas through a and expands them with *tableau*.

```
function $next\mathcal{F}(\mathcal{F},a)$
if $\mathcal{F}$ does not contain a formula $\mathbf{T}\langle a\rangle\alpha$ then return $\emptyset$
else return $tableau(\{\mathbf{T}\alpha|\mathbf{T}\langle a\rangle\alpha \in \mathcal{F}\}$
                $\cup\{\mathbf{F}\alpha|\mathbf{F}\langle a\rangle\alpha \in \mathcal{F}\})$
```

Fig. 1. Function $next\mathcal{F}$

The fields x and f are used to characterize accepting states of the product automaton, and are used to check that all until formulas are fulfilled in a finite number of steps.

If a state s_i of an accepting run ρ contains the until formula $\mathbf{T}\alpha\mathcal{U}^{\mathcal{A}(q)}\beta$, then there must be a state $s_j, i \leq j$ in ρ satisfying the conditions given by the semantics of until. We say that s_j *fulfills* the until formula. If s_i does not fulfill the until formula, then it is possible to show that, according to the axioms of until, s_i contains a formula

```
function $next\_states((\mathcal{F}, w, x, f), a)$
return $\{(\mathcal{F}', w', x', f')$ such that
    $\mathcal{F}' \in next\mathcal{F}(\mathcal{F}, a)$,
    $w' \in nextTSstates(w, a)$,
    $\mathcal{F}' \cup w'$ is consistent,
    if there exist no $\mathbf{T}\langle a\rangle\alpha\mathcal{U}_x^{\mathcal{A}(q)}\beta \in \mathcal{F}$
        then $x' = 1 - x; f' = \checkmark$
        else $x' = x; f' = \downarrow \}$
```

Fig. 2. Function $next\_states$

$\mathbf{T}\langle a_i\rangle\alpha\mathcal{U}^{\mathcal{A}(q')}\beta$, where $q' \in \delta(q, a_i)$[2] and, according to function $next\mathcal{F}(\mathcal{F}_i, a_i)$, s_{i+1} contains a formula $\mathbf{T}\alpha\mathcal{U}^{\mathcal{A}(q')}\beta$. We say that this until formula is *derived* from formula $\mathbf{T}\alpha\mathcal{U}^{\mathcal{A}(q)}\beta$ in state s_i. If a state contains an until formula which is not derived from a predecessor state, we say that the formula is *new*. New until formulas are obtained during the expansion of *tableau*.

In order to check fulfillment of until formulas, we must be able to track them along the states of the run. This is done by using the field x and by extending accordingly signed formulas so that all true until formulas have a label 0 or 1, i.e. they have the form $\mathbf{T}\alpha\mathcal{U}_l^{\mathcal{A}(q)}\beta$ where $l \in \{0, 1\}$. For each state $(\mathcal{F}, w, x, f)$, the label of an until formula in $\mathcal{F}$ is assigned as follows: if it is a derived until formula, then its label is the same as that of the until formula in the predecessor state it derives from, otherwise, if the formula is new, it is given the label $1 - x$.

Function $tableau$ must be suitably modified in order to deal with the labels of until formulas. We assume that it has two parameters: a set of formulas and the value of x.

Let us assume that in a state s_i we have $x = 0$. Then all new until formulas of s_i have label 1, and all until formulas with label 0 must be derived from previous states. If s_i belongs to an accepting run, all until formulas will be fulfilled in a finite number of steps. The value 0 of x is propagated to the next states until a state s_j does not contain any more until formulas with label 0. Then x is switched to 1, and we proceed in the same way. Whenever x changes its value, we set $f = \checkmark$. A state with $f = \checkmark$ is an *accepting state* of the product automaton, and a run ρ containing infinite accepting states is an *accepting run*.

It is an obvious consequence of the construction that:

Proposition 2. *(i) Any accepting run of the product automaton corresponds an infinite path of the transition system (i.e., a temporal answer set of Π) satisfying the initial DLTL formula α; (ii) every infinite path of the transition system which is a model of α corresponds to an accepting run of the product automaton.*

The proof, omitted for lack of space, exploits Theorems 4 and 5 in [17].

Our approach to BMC relies on the well known result [7] that the language accepted by a Büchi automaton is nonempty iff there is a reachable accepting state with a cycle back to itself. The construction of the *(k,l)-loop* is described by the function BMC in Figure 3. The construct **choose in** S returns any of the elements of set S or $null$ if

[2] δ is the transition relation of $\mathcal{A}$.

```
**function** $BMC(max\_k)$
$k := 0$
**do**
    $path :=$ **choose in** $\{s_0 \overset{a_0}{\to} s_1 \overset{a_1}{\to} \dots s_{k+1}$ **such that**
        $s_j \neq s_m$ for $0 \leq j < m \leq k$,
        $s_l = s_{k+1}$ for some $l \leq k$,
        $s_{acc}$ is an accepting state for some $l \leq acc \leq k\}$
    $k := k + 1$
**while** $path = null \wedge k \leq max\_k$
**return** $path$
```

Fig. 3. Function BMC

```
**function** $max()$
$i := 0$
**do**
    $i := i + 1$
    $path :=$ **choose in** $\{s_0 \overset{a_0}{\to} s_1 \overset{a_1}{\to} \dots s_i$ **such that**
        $s_j \neq s_m$ for $0 \leq j < m \leq i\}$
**while** $path \neq null$
**return** $i - 1$
```

Fig. 4. Function max

$S = \emptyset$. With $s_0 \overset{a_0}{\to} s_1 \overset{a_1}{\to} \dots s_i$ we represent a finite path of the product automaton, where s_0 is an initial state and $s_i \in next\_states(s_{i-1}, a_{i-1})$. Given an integer k, we look for a path of length $k + 1$, such that $s_{k+1} = s_l$ for some previous state s_l in the path. Furthermore the loop must contain an accepting state. If such a loop is found, it finitely represents an accepting run. Otherwise, k is increased until max_k is reached.

Observe that the standard approach for bounded model checking in [5] does not guarantee termination, because the path of length k is a path of the transition system, and thus it is not possible to restrict the search to simple paths without missing solutions. On the other hand, we can consider only *simple* paths, that is paths without repeated states. This property allows to define a terminating algorithm, thus achieving *completeness*, by passing the length of the longest simple path as parameter to BMC.

The length of the longest simple path can be found iteratively, searching for a simple path of length i (without loop), and incrementing i at each step (See Figure 4). Since the number of different states if finite, this procedure terminates.

The set of tableau rules can be easily extended to deal with other boolean connectives and derived modal operators. In the following, we use tableau rules for $\Box$ and $\diamond$, using the equivalences $\Box\beta \equiv (\beta \wedge \bigcirc\Box\beta))$ and $\Diamond\beta \equiv (\beta \vee \bigcirc\Diamond\beta))$. Observe that, as false box formulae correspond to negated until formulas, we need to label them with x.

Example 2. Consider the domain description given in Example 1 with the constraints and the property given in Section 2.3. We describe some steps of the (non deterministic) construction of a *(k,l)-loop* for $k = 7$. For the initial state s_0 we have $w_0 = \{\mathbf{T}mail(a), \mathbf{F}mail(b)\}$, $x_0 = 0$, $f_0 = \checkmark$. $\mathcal{F}_0$ contains the following formulas:

$\mathcal{F}_0.1 :\ \mathbf{T}\langle begin\rangle\top$

$\mathcal{F}_0.2 :$ $\mathbf{T}\Box[begin]\langle\mathcal{A}(q_0)\rangle\top$
$\mathcal{F}_0.3 :$ $\mathbf{F}\Box_1(mail(a) \supset \Diamond\neg mail(a))$
$\mathcal{F}_0.4 :$ $\mathbf{T}[begin]\langle\mathcal{A}(q_0)\rangle\top-$ from $\mathcal{F}_0.2$
$\mathcal{F}_0.5 :$ $\mathbf{T}\bigcirc\Box[begin]\langle\mathcal{A}(q_0)\rangle\top$ from $\mathcal{F}_0.2$
$\mathcal{F}_0.6 :$ $\mathbf{F}\bigcirc\Box_1(mail(a) \supset \Diamond\neg mail(a))$ from $\mathcal{F}_0.3$
The first two formulas are the two constraints, where the automaton $\mathcal{A}(q_0)$ is equivalent to the regular program $sense\_mail; (deliver(a) + deliver(b) + wait); begin$ ($\mathcal{A}$ has states $\{q_0, q_1, q_2, q_3\}$, initial state q_0, final state q_3 and transition function $q_1 = \delta(q_0, sense)$, $q_2 = \delta(q_0, del(a)) = \delta(q_0, del(b)) = \delta(q_0, wait)$, $q_3 = \delta(q_2, begin)$). The third formula is the negation of the property. Note that the $\Box$ operator has label 1 since $x_0 = 0$. All other formulas are obtained by applying the $tableau$ rules[3].

Since $\mathcal{F}_0$ contains the formula $\mathbf{T}\langle begin\rangle\top$, we can only execute action $begin$ in s_0. In s_1 we have $w_1 = \{\mathbf{T}mail(a), \mathbf{F}mail(b)\}$, from the domain description, and $x_1 = 1$, $f_0 = \checkmark$. x_1 changes its value from the previous state, because there are no formulas in s_0 with label 0. $\mathcal{F}_1$ is obtained by propagating the "next" formulas in $\mathcal{F}_0$ and by applying *tableau* to them:
$\mathcal{F}_1.1 :$ $\mathbf{T}\langle\mathcal{A}(q_0)\rangle_0\top$ from $\mathcal{F}_0.4$
$\mathcal{F}_1.2 :$ $\mathbf{T}\Box[begin]\langle\mathcal{A}(q_0)\rangle\top$ from $\mathcal{F}_0.5$
$\mathcal{F}_1.3 :$ $\mathbf{F}\Box_1(mail(a) \supset \Diamond\neg mail(a))$ from $\mathcal{F}_0.6$
$\mathcal{F}_1.4 :$ $\mathbf{T}\langle sense\rangle\langle\mathcal{A}(q_1)\rangle_0\top$ from $\mathcal{F}_1.1$
$\mathcal{F}_1.5 :$ $\mathbf{T}[begin]\langle\mathcal{A}(q_0)\rangle\top$ from $\mathcal{F}_1.2$
$\mathcal{F}_1.6 :$ $\mathbf{T}\bigcirc\Box[begin]\langle\mathcal{A}(q_0)\rangle\top$ from $\mathcal{F}_1.2$
$\mathcal{F}_1.7 :$ $\mathbf{F}(mail(a) \supset \Diamond\neg mail(a))$ from $\mathcal{F}_1.3$
$\mathcal{F}_1.8 :$ $\mathbf{F}\neg mail(a))$ from $\mathcal{F}_1.7$
$\mathcal{F}_1.9 :$ $\mathbf{F}\Diamond\neg mail(a))$ from $\mathcal{F}_1.7$
$\mathcal{F}_1.10 :$ $\mathbf{F}\bigcirc\Diamond\neg mail(a))$ from $\mathcal{F}_1.9$
Because of $\mathcal{F}_1.4$ the next action will be $sense$. This action is non deterministic, and we choose $w_2 = \{\mathbf{T}mail(a), \mathbf{T}mail(b)\}$. By continuing with the construction, we can get the following path (we omit the value of the $\mathcal{F}_i$'s in the states, and we write a form $mail(a)$ and b for $mail(b)$):
$(\mathcal{F}_0, \{\mathbf{T}a, \mathbf{F}b\}, 0, \checkmark) \overset{begin}{\rightarrow} (\mathcal{F}_1, \{\mathbf{T}a, \mathbf{F}b\}, 1, \checkmark) \overset{sense}{\rightarrow} (\mathcal{F}_2, \{\mathbf{T}a, \mathbf{T}b\}, 0, \checkmark) \overset{del(b)}{\rightarrow}$
$(\mathcal{F}_3, \{\mathbf{T}a, \mathbf{F}b\}, 0, \downarrow) \overset{begin}{\rightarrow} (\mathcal{F}_4, \{\mathbf{T}a, \mathbf{F}b\}, 0, \downarrow) \overset{sense}{\rightarrow} (\mathcal{F}_5, \{\mathbf{T}a, \mathbf{T}b\}, 1, \checkmark) \overset{del(b)}{\rightarrow}$
$(\mathcal{F}_6, \{\mathbf{T}a, \mathbf{F}b\}, 1, \downarrow) \overset{begin}{\rightarrow} (\mathcal{F}_7, \{\mathbf{T}a, \mathbf{F}b\}, 1, \downarrow) \overset{sense}{\rightarrow} (\mathcal{F}_8, \{\mathbf{T}a, \mathbf{T}b\}, 0, \checkmark)$
Since $\mathcal{F}_8 = \mathcal{F}_2$, the two states n_8 and n_2 are equal. Thus we have an arc back from s_7 to s_2, and the path from s_2 to s_7 contains an accepting state. The path represents a counterexample to the property we wanted to prove.

Let us modify the domain description by adding a fluent $pr(E)$ which associates a priority to the mailboxes. We can add the following rules:
$[deliver(E)]\neg pr(E)$
$[deliver(E)]pr(E') \leftarrow E \neq E', mail(E')$
$[deliver(E)] \perp\leftarrow \neg pr(E), pr(E'), E \neq E'$

[3] For lack of space we consider only the most significant formulas.

By applying function max, we obtain that the longest path has length 17. By executing function $BMC(17)$ we get no solution. Therefore the property $\Box(mail(a) \supset \Diamond\neg mail(a))$ holds in the modified domain description.

5 An ASP Encoding of BMC with Büchi Automata

We now provide a translation into standard ASP of the above procedure for building a path of the product Büchi automaton. We use predicates like `fluent, action, state` to express the type of atoms. As we are interested in infinite runs represented as *(k,l)-loops*, we assume a bound K to the number of states. States are represented in ASP as integers from 0 to K, where K is given by the predicate `laststate(State)`. The predicate `occurs(Action,State)` describes transitions. Occurrence of exactly one action per state can be encoded as:

```
-occurs(A,S):- occurs(A1,S),action(A),action(A1),A!=A1,state(S).
occurs(A,S):- not -occurs(A,S),action(A),state(S).
```

As we have seen, states are associated with a set of fluent literals, a set of signed formulas, and the values of x and f. Fluent literals are represented with the predicate `holds(Fluent,State)`, **T** or **F** formulas with `tt(Formula,State)` or `ff(Formula, State)`, x with the predicate `x(Val,State)` and f with the predicate `acc(State)`, which is true if `State` is an accepting state.

States on the path must be all different, and thus we need to define a predicate `eq(S1,S2)` to check whether the two states $S1$ and $S2$ are equal:

```
eq(S1,S2):- state(S1), state(S2),not diff(S1,S2).
diff(S1,S2):- state(S1),state(S2),tt(F,S1),not tt(F,S2).
diff(S1,S2):- state(S1),state(S2),holds(F,S1),not holds(F,S2).
```

and similarly for other components of a state.

The following constraint requires all states up to K to be different:

```
:-state(S1),state(S2),S1!=S2,eq(S1,S2),laststate(K),S1<=K,S2<=K.
```

Furthermore we need constraints stating that there is a transition from state K to a previous state L[4], and that there is a state S, $L \leq S \leq K$, such that `acc(S)` holds, i.e. S is an accepting state. To do this we compute the successor of state K, and check that it is equal to S.

```
loop(L):- state(L), laststate(K), L<=K,SuccK=K+1, eq(L,SuccK).
accept:- loop(L), state(S), laststate(K), L<=S, S<=K, acc(S).
:- not accept.
```

Given a domain description Π and a set of DLTL formulas $\varphi_1, \dots \varphi_n$, representing constraints or negated properties, we want to compute the temporal answer sets of the domain description Π satisfying the temporal formulas, if any. The rules in Π can be easily translated to ASP, similarly to [14]. In the following we provide the translation of our running example, see [19] for details.

```
action(sense).
action(deliver(a)).
action(deliver(b)).
```

[4] Since states are all different, there will be at most one state equal to the successor of K.

```
action(wait).
fluent(mail(a)).
fluent(mail(b)).
```
action effects:
```
holds(mail(E),NS):- occurs(sense,S), fluent(mail(E)),NS=S+1,
   not -holds(mail(E),NS).
-holds(mail(E),NS):-occurs(deliver(E),S),fluent(mail(E)),NS=S+1.
```
persistence:
```
holds(F,NS):- holds(F,S), fluent(F),NS=S+1,not -holds(F,NS).
-holds(F,NS):- -holds(F,S),fluent(F),NS=S+1,not holds(F,NS).
```
preconditions:
```
:- occurs(deliver(E),S),-holds(mail(E),S).
:- occurs(wait,S), holds(mail(E),S).
```
initial state:
```
-holds(mail(a),0). -holds(mail(b),0).
```

DLTL formulas are represented as ASP terms. In the encoding, each formula $\alpha\mathcal{U}^{\mathcal{A}(q)}\beta$ is represented as `until(A,q,alpha,beta)`, where the automaton $\mathcal{A}$ is described by the predicates `trans(A,Q1,Act,Q2)` defining transitions, and `final(A,Q)` defining final states. Predicate `x(L,S)` gives the value $L = 0, 1$ of x in state S. We introduce the terms `until(A,q,alpha,beta,L)` and `diamond(Act,alpha)` for encoding labeled until formulas and $\langle a\rangle\alpha$ formulas. The expansion of signed formulas can be formulated by means of ASP rules corresponding to the tableau rules given in the previous section.

Disjunction:
```
tt(F1,S) v tt(F2,S):- tt(or(F1,F2),S).
ff(F1,S):- ff(or(F1,F2),S).
ff(F2,S):- ff(or(F1,F2),S).
```
Negation:
```
ff(F,S):- tt(neg(F),S).
tt(F,S):- ff(neg(F),S).
```
Until:
```
tt(until(Aut,Q,F1,F2,1-N),S):- state(S),
   tt(until(Aut,Q,F1,F2),S),x(N,S),label(N).
tt(or(F2,and(F1,diamond(Act,until(Aut,Q1,F1,F2,L)))),S):-
   tt(until(Aut,Q,F1,F2,L),S),state(S),label(L),final(Aut,Q),
   occurs(Act,S),choose(until(Aut,Q,F1,F2,L),S,Act,Q1).
tt(and(F1,diamond(Act,until(Aut,Q1,F1,F2,L))),S):- state(S),
   tt(until(Aut,Q,F1,F2,L),S),label(L),not final(Aut,Q),
   occurs(Act,S),choose(until(Aut,Q,F1,F2,L),S,Act,Q1).
ff(F2,S):- state(S),ff(until(Aut,Q,F1,F2),S), final(Aut,Q).
ff(diamond(Act,until(Aut,Q1,F1,F2)),S):-state(S),
   occurs(Act,S),ff(until(Aut,Q,F1,F2),S),trans(Aut,Q,Act,Q1).
```
Diamond
```
tt(F,NS):- tt(diamond(Act,F),S), NS=S+1.
ff(F,NS):- ff(diamond(Act,F),S),occurs(Act,S), NS=S+1.
```

Note that, to express splitting of sets of formulas, as in the case of disjunction, we can exploit disjunction in the head of clauses, provided by some ASP languages such as DLV, or choice constructs available in other languages. The predicate `choose` below non deterministically chooses a transition Q1 among those possible for action `Act` in the automaton `Aut`, and uses that choice in the expansion of the until formula:

```
choose(until(Aut,Q,F1,F2,L),S,Act,Q1):- state(S),action(Act),
  not-choose(until(Aut,Q,F1,F2,L),S,Act,Q1),trans(Aut,Q,Act,Q1).
-choose(until(Aut,Q,F1,F2,L),S,Act,Q1):- state(S),action(Act),
  choose(until(Aut,Q,F1,F2,L),S,Act,Q2),Q1!=Q2.
```

Inconsistency of signed formulas is formulated with the following constraints:

```
:- ff(true,S), state(S).
:- tt(F,S), ff(F,S), state(S).
:- tt(diamond(Act1,F),S),tt(diamond(Act2,F),S), Act1!=Act2.
:- tt(F,S), not holds(F,S).
:- ff(F,S), not -holds(F,S).
```

As a difference with the tableau construction, rather than introducing the translation of formula $\mathbf{T} \bigvee_{a \in \Sigma} \langle a \rangle \top$ in the initial state, we include the rule

```
   tt(diamond(A,true),S):- occurs(A,S).
```

as we know that at least one action (and at most one) occurs in a state.

Predicates x and acc are defined as follows:

```
cont(S):-state(S),x(Lab,S),tt(diamond(_,until(_,_,__,Lab)),S).
x(Lab,SN):- x(Lab,S),SN=S+1, cont(S).
-acc(SN):- x(Lab,S),SN=S+1, cont(S).
x(1-Lab,SN):- x(Lab,S),SN=S+1, not cont(S).
acc(SN):- x(Lab,S),SN=S+1, not cont(S).
x(0,0). acc(0).
```

Finally, we must add a fact `tt(`$tr(\varphi_i)$`,0)` for each DLTL formula φ_i to be satisfied in the model, where $tr(\varphi_i)$ is the ASP term representing φ_i.

It is easy to see that the (groundization of the) encoding in ASP is linear in the size of the formula ϕ to be verified and in the number f of ground fluents while quadratic in the size of k. We can prove that there is a one to one correspondence between the extensions of a domain description satisfying a given temporal formula and the answer sets of the ASP program encoding the domain and the formula.

Proposition 3. *Let Π be a domain description whose temporal answer sets are total, let $tr(\Pi)$ be the ASP encoding of Π (for a given k), and let ϕ be a DLTL formula.*

If there is a temporal answer set of Π that satisfies the formula ϕ, then there exists an answer sets of the ASP program $tr(\Pi) \cup tt(tr(\phi), 0)$ (where $tr(\phi)$ is the ASP term representing ϕ); and vice versa.

For achieving completeness, the search for the longest simple path can be done by removing from the above ASP encoding the rules for defining loops and the rules for defining the Büchi acceptance condition.

The translation has been run in iClingo [13]. For the dining philosophers problems in [23], the scalability of the approach in this paper is similar to the one for the method (without Büchi automaton) in [19] and the one in [23], when looking for a counterexample. E.g., a counterexample for DP(12) is found in 183 seconds, wrt 274 seconds for a Clingo implementation of the method in [19] — see also Appendix C in that paper. The search for the longest simple path is substantially more costly and practically feasible only for problems where the action domain is sufficiently constrained.

6 Conclusions

The paper presents a bounded model checking approach for the verification of properties of temporal action theories in ASP. The temporal action theory is formulated in a temporal extension of ASP, where DLTL constraints in domain descriptions allow for state trajectory constraints to be captured. The approach provides a uniform ASP metodology for specifying and verifying domain descriptions, which can be used for several reasoning tasks, including business process verification [10] and planning.

Helianko and Niemelä [23] developed a compact encoding of bounded model checking of LTL formulas as the problem of finding stable models of logic programs. In [19] this encoding is extended to address the verification of action domains including DLTL constraints. In this paper, we follow a different approach to BMC which exploits the Büchi automaton construction to achieve completeness. [8] first proposed the use of the Büchi automaton in BMC. As a difference, our encoding in ASP is defined without assuming that the Büchi automaton is computed in advance.

The action language in this paper is related to the logic programming based planning language $\mathcal{K}$ [12] and with the languages $\mathcal{C}$ and $\mathcal{C}^+$ [21, 20]. Unlike $\mathcal{K}$, $\mathcal{C}$ and $\mathcal{C}^+$ our action language does not allow for concurrent actions, but it provides general temporal constraints. $\mathcal{K}$, $\mathcal{C}$ and $\mathcal{C}^+$ can perform several kinds of reasoning, such as, prediction, postdiction and planning. However, they do not exploit standard temporal logic constructs to reason about actions.

$\mathcal{ESG}$ [9] is a second order extension of CTL* for reasoning about nonterminating Golog programs. The paper presents a method for verification of a first order CTL fragment of $\mathcal{ESG}$, using model checking and regression based reasoning. Because of first order quantification, this fragment is in general undecidable.

In [1] the verification problem for action logic programs with nonterminating behavior is addressed using an action formalism based on a temporalized description logic, $\mathcal{ALCO}$-LTL. DLTL does not allow for first order constructs as $\mathcal{ALCO}$-LTL, while it allows for the specification of regular expressions.

In [6] Cabalar introduces normal forms for Temporal Equilibrium Logic (TEL), an extension of the Answer Set semantics to arbirary theories in the syntax of Linear Temporal Logic. The rules in Π, in our action theories, appear to be in normal form. It would be interesting to investigate the possibility of mapping the LTL fragment of our action theories into TEL.

Acknowledgments
We thank the anonymous referees for their helpful comments. This work has been partially supported by the project of Regione Piemonte "ICT4Law".

References

1. Franz Baader, Hongkai Liu, and Anees ul Mehdi. Verifying properties of infinite sequences of description logic actions. In *ECAI*, pages 53–58, 2010.

2. F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2):123–191, 2000.
3. Jorge A. Baier, Fahiem Bacchus, and Sheila A. McIlraith. A heuristic search approach to planning with temporally extended preferences. *Artif. Intell.*, 173(5-6):593–618, 2009.
4. C. Baral and J. Zhao. Non-monotonic temporal logics for goal specification. In *IJCAI 2007*, pages 236–242, 2007.
5. A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.
6. Pedro Cabalar. A normal form for linear temporal equilibrium logic. In *JELIA, LNCS 6341*, pages 64–76, 2010.
7. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 2001.
8. E.M. Clarke, D. Kroening, J. Ouaknine, and O. Strichman. Completeness and complexity of bounded model checking. In *VMCAI*, pages 85–96, 2004.
9. J. Claßen and G. Lakemeyer. A logic for non-terminating Golog programs. In *Proc. KR 2008*, pages 589–599, 2008.
10. D. D'Aprile, L. Giordano, V. Gliozzi, A. Martelli, G.L. Pozzato, and D. Theseider Dupré. Verifying business process compliance by reasoning about actions. In *CLIMA 2010, LNCS 6245*, 2010.
11. Giuseppe De Giacomo, Fabio Patrizi, and Sebastian Sardiña. Generalized planning with loops under strong fairness constraints. In *Proc. KR 2010*, 2010.
12. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A logic programming approach to knowledge-state planning: Semantics and complexity. *ACM TOCL*, 5(2):206–263, 2004.
13. M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. Engineering an incremental ASP solver. In *Proc. ICLP08*, volume 5366 of *LNCS*, pages 190–205, 2008.
14. M. Gelfond. *Handbook of Knowledge Representation, ch. 7, Answer Sets*. Elsevier, 2007.
15. A. Gerevini and D. Long. Plan constraints and preferences in PDDL3. *Technical Report, Department of Electronics and Automation, University of Brescia, Italy*, 2005.
16. R. Gerth, D. Peled, M.Y.Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *15th Work. Protocol Specification, Testing and Verification*, 1995.
17. L. Giordano and A. Martelli. Tableau-based automata construction for dynamic linear time temporal logic. *Annals of Mathematics and AI*, 46(3):289–315, 2006.
18. L. Giordano, A. Martelli, and C. Schwind. Specifying and verifying interaction protocols in a temporal action logic. *Journal of Applied Logic*, 5:214–234, 2007.
19. L. Giordano, A. Martelli, and D. Theseider Dupré. Reasoning about actions with temporal answer sets. *TPLP*, To appear. Available at http://arxiv.org/abs/1110.3672.
20. E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, , and H. Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153(1-2):49–104, 2004.
21. E. Giunchiglia and V. Lifschitz. An action language based on causal explanation: Preliminary report. In *AAAI/IAAI*, pages 623–630, 1998.
22. F. Giunchiglia and P. Traverso. Planning as model checking. In *Proc. The 5th European Conf. on Planning (ECP'99)*, pages 1–20, 1999.
23. K. Heljanko and I. Niemelä. Bounded LTL model checking with stable models. *TPLP*, 3(4-5):519–550, 2003.
24. J.G. Henriksen and P.S. Thiagarajan. Dynamic linear time temporal logic. *Annals of Pure and Applied logic*, 96(1-3):187–207, 1999.
25. M. Pistore and P. Traverso. Planning as model checking for extended goals in non-deterministic domains. In *Proc. IJCAI 2001*, pages 479–486, 2001.
26. S. Sohrabi and S. A. McIlraith. Optimizing web service composition while enforcing regulations. In *ISWC 2009, Chantilly, USA, LNCS 5823*, pages 601–617, 2009.