Temporally-Expressive Planning as Constraint Satisfaction Problems

Yuxiao Hu
Department of Computer Science
University of Toronto
Toronto, Ontario M5S 3G4, Canada
yuxiao (a) cs toronto edu

Abstract

Due to its important practical applications, temporal planning is of great research interest in artificial intelligence. Yet most of the work in this area so far is limited in at least two ways: it only considers temporally simple domains and it has restricted decision epochs as the potential happening time of actions. Because of these simplifying assumptions, existing temporal planners are in fact not complete.

In this paper, we focus on these limitations, and propose an alternative view of temporal planning by investigating a new declarative semantics of PDDL. We then show a natural encoding of this semantics in a constraint programming setting. It turns out that this encoding unifies planning and scheduling, and captures most of the temporal expressiveness of PDDL. The resulting CSP-based temporal planner can solve more general planning problems than the current *state-of-the-art*.

Introduction

In recent years, much work has been done in temporal planning, *e.g.* (Smith & Weld 1999; Bacchus & Ady 2001; Vidal & Geffner 2004; Chen, Wah, & Hsu 2006). However, as Cushing *et al.* (2007) recently show, most of them only deal with a limited subset of all temporal planning problems. Typically, they make the following assumptions:

1. Actions can only have temporally annotated preconditions and effects in a restricted form, *e.g.* preconditions always hold over the whole duration of an action, and effects only take place at the end;
2. Actions may only happen at certain decision epochs, *e.g.* immediately after some other action is starting or ending.

These restrictions simplify the planning problem, but unfortunately, they also render the planners incomplete, as they cannot find plans for temporally expressive problems, such as the ones shown in Figure 1, even though plans obviously do exist (Cushing *et al.* 2007).

PDDL (Fox & Long 2003; Edelkamp & Hoffmann 2004), on the other hand, offers much more expressiveness for describing temporal planning domains and problems, making it possible to express that an action's precondition must hold at the start, at the end or within the (open) interval of the duration, and that its effect must take place at the start or at the end of the action. Moreover, there is no restriction on the happening time of actions, since an action can happen at any positive time point, provided that its preconditions are satisfied and no mutex violation arises.

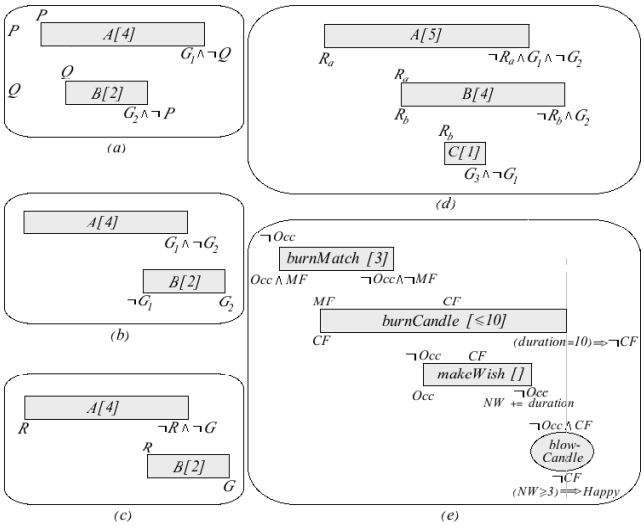

Figure 1: **"Temporally expressive" planning problems.** Examples (a)–(d) are taken from (Cushing *et al.* 2007) and (e) will be our running example. Numbers in brackets after the action names indicate the duration constraint; preconditions and effects are labeled above and below the actions, respectively, at the time points they are enforced.

As a running example, the "happy birthday-boy" domain in Figure 1(e) graphically represents a temporal PDDL problem, where the goal is to become "*Happy*". It has three durative actions and a simple (instantaneous) action. Intuitively, burning a match, lasting for 3 time units, requires the birthday-boy to not be occupied (*Occ*) with other things, and as an effect, he is occupied and has match flame (MF) throughout the duration. Burning a candle needs the match flame to start with, and lasts for *at most* 10 time units; candle fire (CF) is produced at the start, and stays all through the duration of this action, but if the burning time reaches 10

time units, the fire is out automatically. Making a wish is an action with arbitrary duration, requiring his non-occupancy at the start and candle fire within the duration; as an effect, the number of wishes (NW) is increased by the number of time units the action spans. Finally, the simple action of blowing out the candle requires that candle fire exist, and the effects include, apart from putting out the fire, becoming happy if he has successfully made no less than 3 wishes.

This toy example has "required concurrency" (Cushing *et al.* 2007), since burning a candle must start while the match is burning, and making a wish must be contained in the candle-burning process. Beyond the notion of temporal expressiveness of Cushing *et al.*, it also illustrates the use of duration inequality (*e.g.* "≤ 10") and duration-related effects (*e.g.* number of wishes depends on the duration). All these features contribute to prevent most existing PDDL planners from finding a solution to the problem.

A gap thus exists between the potential breadth of problems that PDDL can represent and the actual narrowness that existing PDDL planners can solve. This gap may seem inevitable at first glance, but in the rest of this paper, we shall show that this is not the case.

We do so by first investigating a declarative semantics for the temporal subset of PDDL, recently proposed by Claßen, Hu and Lakemeyer (2007). Based on this, we then present a way to directly encode the semantics in the language of constraint programming. Finally we show that the resulting CSP-based planner is general and powerful enough to solve planning problems in temporally expressive domains that almost no other existing PDDL planner is able to solve.

The Temporal Subset of PDDL

The target subset of PDDL we discuss here includes all features in PDDL 2.1 (Fox & Long 2003) and 2.2 (Edelkamp & Hoffmann 2004), except for continuous effects and derived predicates. Following (Claßen, Hu, & Lakemeyer 2007), instead of using the somewhat clumsy LISP-like syntax, we appeal to the following more compact representation. A PDDL planning problem $\mathcal{P}$ consists of[1]

1. A finite set of types $\tau_1, \cdots, \tau_l$;
2. A finite set of predicates F_j with arguments $\vec{x_j}$ of type $\vec{\tau_j}$, *e.g. Occ*, MF, *Happy*;
3. A finite set of functions f_j with arguments $\vec{x_j}$ of type $\vec{\tau_j}$, *e.g.* NW;
4. A finite set of typed objects $o_1 : \tau_{o_1}, \cdots, o_k : \tau_{o_k}$;
5. A finite set of operators A_i (described below);
6. A finite set of timed initial literals $\langle t_1, L_1 \rangle, \cdots, \langle t_r, L_r \rangle$, where t_i is a number and L_i is a ground literal (predicate instance or its negation).
7. The initial state description, consisting of a finite collection of literals and formulas of the form $f(\vec{o}) = c$ where c is a number, *e.g. HasCake*, $NW = 0$;
8. A goal description $\mathcal{G}$, which is a precondition formula, *e.g. Happy* $\wedge \neg$*Occ*.

[1] Our running example does not use all features introduced here, *e.g.* objects, typing, arguments to fluents and actions, *etc*.

There are two kinds of action operators: simple ones and durative ones. A *simple action* A is represented by a triple $\langle \vec{z} : \vec{\tau}, \pi_A, \epsilon_A \rangle$, where $\vec{z}$ are A's arguments and $\vec{\tau}$ their corresponding types, π_A is its precondition, and ϵ_A is its effect, the latter being a conjunction of conditional effects of the form

$$\begin{array}{lcl} \forall \vec{x_j} : \vec{\tau_j} . \gamma^{+}_{F_j,A}(\vec{x_j}, \vec{z}) & \Rightarrow & F_j(\vec{x_j}) \\ \forall \vec{x_j} : \vec{\tau_j} . \gamma^{-}_{F_j,A}(\vec{x_j}, \vec{z}) & \Rightarrow & \neg F_j(\vec{x_j}) \\ \forall \vec{x_j} : \vec{\tau_j} . \gamma^{v}_{f_j,A}(\vec{x_j}, y_j, \vec{z}) & \Rightarrow & f_j(\vec{x_j}) = y_j \end{array} \quad (1)$$

Here, $\gamma \Rightarrow \psi$ means that if γ holds before the action, then ψ holds afterwards. Without loss of generality, we assume that there is at most one single effect of each form for any given F_j or f_j in ϵ_A, and that there is at most one value for y_j to satisfy $\gamma^{v}_{f_j,A}(\vec{x_j}, y_j, \vec{z})$ for any instance of $\vec{x_j}$ and $\vec{z}$. For example, the "*blowCandle*" action is represented by

$$\langle \{\},\ \neg \mathit{Occ} \wedge CF,\ [\textsc{True} \Rightarrow \neg CF] \wedge [(NW \geq 3) \Rightarrow \mathit{Happy}] \rangle$$

A *durative action* A is represented by a four-tuple $\langle \vec{z} : \vec{\tau}, \delta_A, \pi_A, \epsilon_A \rangle$, where $\vec{z} : \vec{\tau}$ is the same as before, and

- $\delta_A = \langle \delta^s_A, \delta^e_A \rangle$ where δ^s_A and δ^e_A are the start and end duration constraints, respectively, each being an (in)equality between arithmetic formulas whose numerical terms are either numbers, functions or the special variable *duration*.
- $\pi_A = \langle \pi^s_A, \pi^o_A, \pi^e_A \rangle$ are the preconditions at the start, during the *open* interval and at the end of A, respectively.
- $\epsilon_A = \langle \epsilon^s_A, \epsilon^e_A \rangle$ are the start and end effects. Start effects are represented by a conjunction of conditional effects in the form of (1); End effects are a conjunction of conditional effects of the form $\langle \gamma^s, \gamma^o, \gamma^e \rangle \Rightarrow \psi$, where γ^s, γ^o and γ^e are the start, overall and end premises, respectively. ψ takes place only if all premises hold at their corresponding time points.

For example, "*burnCandle*" is represented by

$$\Big\langle \{\},\ \mathit{duration} \leq 10,\ \langle MF, CF, \textsc{True} \rangle, \\ \big\langle [\textsc{True} \Rightarrow CF], [\langle \textsc{True}, \textsc{True}, \mathit{duration} = 10 \rangle \Rightarrow \neg CF] \big\rangle \Big\rangle$$

Declarative Semantics for Temporal PDDL

Lifschitz (1986) was the first to define a state-transitional semantics for STRIPS. PDDL (McDermott & the AIPS'98 Planning Competition Committee 1998) did not have a formal semantics until Fox and Long (2003) extended Lifschitz' definition to their PDDL 2.1. Later versions of PDDL adapted this semantics for the new features (Edelkamp & Hoffmann 2004; Gerevini & Long 2005).

However, this state-transitional semantics is purely metatheoretic, in that it is not directly clear how it may help understand a planning problem in terms of logic theories. The latter is interesting to us, since it allows us to logically analyze and encode the dynamics of planning problems. As a result, a declarative semantics is also highly desirable.

The pioneering work on defining a logic-based semantics for planning languages owes to Lin and Reiter (1997), who showed that state updates in relational STRIPS is equivalent to first order progression of the corresponding basic action theory (BAT) in the situation calculus (Reiter 2001).

Claßen *et al.* (2007) extended this result to the ADL subset of PDDL. Recently, Claßen, Hu and Lakemeyer (2007) further added the temporal features, so that the declarative semantics captures roughly PDDL 2.1 plus timed-initial literals.

In the rest of this section, we shall briefly review the semantic definition in (Claßen, Hu, & Lakemeyer 2007) (*c.f.* the literature for technical details), identify its limitations, and propose some modifications to it to solve the problems.

$\mathcal{ES}$-Based Semantics for PDDL

In (Claßen, Hu, & Lakemeyer 2007), the semantics is defined in terms of BATs in the logic $\mathcal{ES}$ (Lakemeyer & Levesque 2004), a first-order modal logic for reasoning about dynamic domains. The two modal operators are $\Box$ and $[\,]$, with $\Box\alpha$ meaning that α always holds and $[a]\alpha$ meaning that α holds after action a. As a convention, the last parameter to an action term represents the happening time, and $\Box \mathit{time}\big(A(\vec{x},t)\big) = t$ extracts it. The BAT is defined as

$$\Sigma = \Sigma_0 \cup \Sigma_{pre} \cup \Sigma_{post} \tag{2}$$

where Σ_0 is the initial database consisting of a finite set of fluent sentences.[2] Σ_{pre} is a precondition axiom of the form[3]

$$\Box \mathit{Poss}(a) \equiv \pi \tag{3}$$

where π is a fluent formula with a being the only free variable. For example,[4]

$$\Box \mathit{Poss}(move(x,y,t)) \supset \mathit{Clear}(x) \wedge \mathit{Clear}(y)$$

says that in any situation, it is possible to move x onto y at time t only if both x and y are clear on the top.

Σ_{post} is a finite set of successor state axioms (SSAs) of the form (4) for predicates and (5) for functions.

$$\Box[a]F(\vec{x}) \equiv \gamma_F(\vec{x}) \tag{4}$$

$$\Box[a]f(\vec{x}) = y \equiv \gamma_f(\vec{x},y) \tag{5}$$

For example,

$$\Box[a]\,On(x,y) \equiv \exists t.\, a = move(x,y,t) \vee On(x,y) \wedge \neg\exists z,t.\, a = move(x,z,t) \tag{6}$$

says that in any situation, x is directly on y after action a if and only if a is an action to move x onto y, or x was on y and a is not an action that moves it elsewhere.

Each durative action a is represented by its start event $\mathit{start}(a,t)$ and end event $\mathit{end}(a,t)$, denoting the corresponding happenings at time t. A fluent predicate $\mathit{Performing}(a)$ is used to remember whether a is in progress (started but not yet ended), and a fluent function $\mathit{since}(a)$ stores the happening time of a's start event. They are captured by the SSAs

$$\Box[a]\mathit{Performing}(a') \equiv \exists t.\, a = \mathit{start}(a',t) \vee \mathit{Performing}(a') \wedge \forall t.\, a \neq \mathit{end}(a',t) \tag{7}$$

$$\Box[a]\mathit{since}(a') = t \equiv a = \mathit{start}(a',t) \vee \mathit{since}(a') = t \wedge \forall t.\, a \neq \mathit{start}(a',t) \tag{8}$$

Additionally, the following precondition axioms ensure that a's start event can happen only when a is not running, and its end event can occur only when it is in progress:

$$\Box \mathit{Poss}(\mathit{start}(a,t)) \supset \neg \mathit{Performing}(a) \tag{9}$$

$$\Box \mathit{Poss}(\mathit{end}(a,t)) \supset \mathit{Performing}(a) \tag{10}$$

Finally, they define the notion of executability with the special predicate *Executable* as

$$\mathit{Executable} \tag{11}$$

$$\Box[a]\mathit{Executable} \equiv \mathit{Executable} \wedge \mathit{Poss}(a) \tag{12}$$

(11), belonging to Σ_0, says that the initial situation is executable (reachable through an executable plan); (12) in Σ_{post} inductively ensures that a situation is executable if and only if the sequence of actions is.

With these logical foundations, they define the mapping from PDDL to BAT as follows.

The ADL subset The cases for Σ_0 and Σ_{pre} are straightforward and thus omitted here. As for Σ_{post}, the SSAs for predicates are obtained by extracting the positive and negative conditions for each predicate F_j as

$$\gamma^+_{F_j} \stackrel{def}{=} \bigvee_{F_j(\vec{x_j}) \text{ effect in } \epsilon_{A_i}} \exists \vec{z_i}, t.\, a = A_i(\vec{z_i},t) \wedge \gamma^+_{F_j,A_i} \tag{13}$$

$$\gamma^-_{F_j} \stackrel{def}{=} \bigvee_{\neg F_j(\vec{x_j}) \text{ effect in } \epsilon_{A_i}} \exists \vec{z_i}, t.\, a = A_i(\vec{z_i},t) \wedge \gamma^-_{F_j,A_i} \tag{14}$$

and then integrating them into an SSA

$$\Box[a]F_j(\vec{x_j}) \equiv \gamma^+_{F_j} \wedge \tau_{\vec{F_j}}(\vec{x_j}) \vee F_j(\vec{x_j}) \wedge \neg\gamma^-_{F_j} \tag{15}$$

where $\tau_{\vec{F_j}}(\vec{x_j})$ are typing constraints ensuring that only instances with correct argument types can become true.

Functional fluents The SSAs for functional fluents f_j are constructed similarly by first extracting the update condition

$$\gamma^v_{f_j} \stackrel{def}{=} \bigvee_{f_j(\vec{x_j}) \text{ effect in } \epsilon_{A_i}} \exists \vec{z_i}, t.\, a = A_i(\vec{z_i},t) \wedge \gamma^v_{f_j,A_i} \tag{16}$$

and then integrating it into the SSA[5]

$$\Box[a]f_j(\vec{x_j}) = y_j \equiv \gamma^v_{f_j} \wedge \tau_{\vec{f_j}}(\vec{x_j}) \vee f_j(\vec{x_j}) = y_j \wedge \neg\exists y'.\, (\gamma^v_{f_j})^{y_j}_{y'} \tag{17}$$

Durative actions For durative actions, the start preconditions π^s_A and effects ϵ^s_A of a PDDL durative action A are mapped to the simple action $\mathit{start}(A,t)$ in the BAT. Similarly, π^e_A and ϵ^e_A are mapped to $\mathit{end}(A,t)$.

For the end duration constraint δ^e_A, the duration of action A is obtained by $(t - \mathit{since}(A))$ when $\mathit{end}(A,t)$ is about to happen, so δ^e_A is ensured by the precondition axiom

$$\Box \mathit{Poss}(\mathit{end}(A,t)) \supset (\delta^e_A)^{\mathit{duration}}_{t-\mathit{since}(A)} \tag{18}$$

[2] A fluent formula is one with no $\Box$, $[\,]$ and the special predicate *Poss*; a fluent sentence is a fluent formula without free variables.

[3] $[\,]$ has a higher priority than logical connectives, and $\Box$ has a lower priority than logical connectives.

[4] In BAT, precondition axioms for primitive actions are of the form $\Box\mathit{Poss}(a) \equiv \pi$; in this paper, "$\supset$" is used in many places, and "$\equiv$" can be obtained by making a completeness assumption.

[5] In this paper, $(\alpha)^{x_1\cdots x_m}_{o_1\cdots o_m}$ denotes the formula obtained by simultaneously replacing all free occurrences of x_i in α with o_i.

The test of start duration constraint δ_A^s is postponed to the end event when the duration becomes available. For this purpose, a functional fluent f_i^s is introduced for each functional fluent f_i appearing in δ_A^s, so that f_i^s stores the value of f_i at the start, and can be used at the end to test the satisfiability of δ_A^s. Formally, this is realized by the following SSA for f_i^s and an additional precondition for $\mathit{end}(A,t)$:

$$\Box[a]f_i^s(\vec{x}) = y \equiv \exists t.\, a = \mathit{start}(A,t) \wedge y = f_i(\vec{x}) \vee f_i^s(\vec{x}) = y \wedge \neg\exists t.\, a = \mathit{start}(A,t) \quad (19)$$

$$\Box \mathit{Poss}(\mathit{end}(A,t)) \supset (\delta_A^s)_{f_i^s(\vec{x})\ t-\mathit{since}(A)}^{f_i(\vec{x})\ \mathit{duration}} \quad (20)$$

In domains with only discretized action effects, invariant conditions can be protected by disallowing actions to happen, if their effects will violate the invariant condition of running durative actions. Formally, this is achieved by

$$\Box \mathit{Poss}(a) \supset \bigwedge_{a' \in \mathcal{A}} \mathcal{R}\big[a, \big(\mathit{Performing}(a') \supset \pi_{a'}^o\big)\big] \quad (21)$$

where $\mathcal{A}$ is the set of all durative actions, $\pi_{a'}^o$ denotes the invariant condition of a', and $\mathcal{R}[a,\phi]$ is the regressed formula of ϕ through a (Lakemeyer & Levesque 2004).

Finally, for each inter-temporal conditional effect of action A_j of the form $\forall \vec{x_i} : \vec{\tau_i}.\, \langle \gamma_i^s, \gamma_i^o, \gamma_i^e \rangle \Rightarrow \psi_i$, new fluent predicates $C_i^s(\vec{z_j}, \vec{x_i})$ and $C_i^o(\vec{z_j}, \vec{x_i})$ are introduced, so as to remember the truth values of the premises, with SSAs

$$\Box[a]C_i^s(\vec{z_j}, \vec{x_i}) \equiv \exists t.\, a = \mathit{start}(A_j(\vec{z_j}, t)) \wedge \gamma_i^s \vee C_i^s(\vec{z_j}, \vec{x_i}) \wedge \neg\exists t.\, a = \mathit{start}(A_j(\vec{z_j}), t) \quad (22)$$

$$\Box[a]C_i^o(\vec{z_j}, \vec{x_i}) \equiv \exists t.\, a = \mathit{start}(A_j(\vec{z_j}), t) \wedge \mathcal{R}[a, \gamma_i^o] \vee C_i^s(\vec{z_j}, \vec{x_i}) \wedge \neg\exists t.\, a = \mathit{start}(A_j(\vec{z_j}), t) \wedge \mathcal{R}[a, \gamma_i^o] \quad (23)$$

such that the effect becomes a local one of $\mathit{end}(A_j, t)$:

$$\forall \vec{x_i} : \vec{\tau_i}.\, C_i^s(\vec{z_j}, \vec{x_i}) \wedge C_i^o(\vec{z_j}, \vec{x_i}) \wedge \gamma_i^e \Rightarrow \psi_i \quad (24)$$

(24) is an instance of (1), so the construction of SSA for ψ_i follows from (13)–(17).

Incorporating True Concurrency

One limitation of the semantic definition above is that it is based on $\mathcal{ES}$, which only supports interleaved concurrency. This mean that although actions may literally happen at the same time, there is actually an ordering among them. This notion of concurrency is weaker than the standard semantics of PDDL 2.1. So in this section, we present some modifications to their definition to incorporate true concurrency.

We base the new semantics upon BATs in the concurrent temporal situation calculus (Chapter 7 of (Reiter 2001)), but using a syntax similar to that of $\mathcal{ES}$. Syntactically, the biggest change is that the $[\,]$ operator has a set of actions, instead of a single action, as its parameter, and the parameter to *Poss* may be either a set or a single action. For example,

$$[\{turnKnob(t), push(t)\}](DoorOpen \wedge \mathit{Poss}(\{goOut(t')\}))$$

says that if we turn the knob and push the door simultaneously at time t in the initial situation, the door opens and it is possible to go out at time t'.

In the concurrent temporal situation calculus, the BAT is

$$\Sigma = \Sigma_0 \cup \Sigma_{pre} \cup \Sigma_{post} \cup \Sigma_{una} \cup \mathcal{FA}$$

where

- Σ_0 is the same as before;
- Σ_{pre} is like before, except that Axiom (21) is adapted to[6]

$$\Box \mathit{Poss}(c) \supset \bigwedge_{a \in \mathcal{A}} \mathcal{R}\big[c, \big(\mathit{Performing}(a) \supset \pi_a^o\big)\big] \quad (25)$$

- Σ_{post} is like before, except that in each SSA, $[a]$ is replaced by $[c]$ to denote a set of actions, all sub-formulas of the form $a = A_i$ are replaced by $A_i \in c$, and $\mathcal{R}[a,\phi]$ by $\mathcal{R}[c,\phi]$. In the rest of this paper, when we refer to SSAs, we always mean those adapted to true concurrency with this substitution. For example, Axiom (6) becomes

$$\Box[c]\,On(x,y) \equiv \exists t.\, move(x,y,t) \in c \vee On(x,y) \wedge \neg\exists z, t.\, move(x,z,t) \in c \quad (26)$$

- Σ_{una} is a set of unique names axioms for constants and action names.
- $\mathcal{FA}$ is a set of foundational axioms (c.f. Chapter 7 of (Reiter 2001) for details), including (but not limited to)

$$\Box \mathit{Poss}(c) \supset \exists a.\, a \in c \quad (27)$$

saying that a concurrent happening must contain at least one action, and

$$\Box \mathit{Poss}(c) \supset \exists t.\, \big(\forall a.\, a \in c \supset time(a) = t\big) \quad (28)$$

saying that actions in a concurrent step happen at the same time. A unique functional fluent *now* is used to represent the time of the most recent happening[7] with the SSA

$$\Box[c]\mathit{now} = \mathit{time}(c) \quad (29)$$

Finally, the following axiom in $\mathcal{FA}$ ensures that actions happen chronologically:

$$\Box \mathit{Poss}(c) \supset \mathit{now} < \mathit{time}(c) \quad (30)$$

Timed initial literals

In (Claßen, Hu, & Lakemeyer 2007), timed initial literals (TILs) are handled with "obligatory actions," which needs extra mechanism to implement in practice. Here, we use flag predicates to simulate the coerciveness of these actions, and thus realize the timed initial literals.

To be specific, for each timed initial literal $\langle t_k, L_k \rangle$, we add a new and unique asserting action $A_k(t)$ along with a flag predicate Achvd_k. $A_k(t)$ can only happen at time t_k:

$$\Box \mathit{Poss}(A_k(t)) \supset t = t_k \quad (31)$$

The only effect of action $A_k(t)$ is to make L_k and Achvd_k hold, whereas the flag Achvd_k, initialized to FALSE, indicates whether the asserting action $A_k(t)$ is executed:

$$\Box[c]\mathit{Achvd}_k \equiv \exists t.\, A_k(t) \in c \vee \mathit{Achvd}_k \quad (32)$$

To enforce the timed initial literals, we need the axiom

$$\Box \mathit{Poss}(c) \supset \big(\forall k.\, (t_k \leq \mathit{time}(c)) \supset \mathit{Achvd}_k\big) \quad (33)$$

This forces all the asserting actions that should happen before the concurrent actions c to be executed, and thus the corresponding timed initial literals asserted.

[6] Without going into detail, $\mathcal{R}[c,\phi]$ denotes the regressed formula of ϕ through the set of concurrent actions c (Reiter 2001).

[7] Equivalently, *now* is the starting time of the current situation. In Reiter's formalism, this is denoted by the function $\mathit{start}(s)$.

Correctness and Remarks

We have the following correctness result for the declarative semantics defined in this section.

Theorem 1. *Let Σ be the basic action theory obtained from a PDDL problem $\mathcal{P}$ and $\mathcal{G}$ be the goal formula. Further let $S = \{c_1, \cdots, c_n\}$ be a sequence of ground instantaneous concurrent actions. Then the corresponding PDDL plan*[8] *of S is valid if and only if*

$$\Sigma \models [c_1] \cdots [c_n](\textit{Executable} \wedge \mathcal{G} \wedge \neg\exists a.\, \textit{Performing}(a))$$

Proof. (Sketch)
The correctness for a variant of this theorem in the interleaved-concurrency setting is given in (Hu 2006). The idea of proof here is similar. We introduce auxiliary predicates and functions in the PDDL problem definition to characterize the process-related properties, which correspond to the auxiliary fluents in the BAT, like *Performing*(a), *since*(a), C_i, *etc.* Then we show that the PDDL state update due to each happening, including the update of the process-related properties, corresponds to the progression of the BAT through the concurrent actions. □

The nice property of the declarative semantics introduced here is that it breaks durative actions down into simple ones, such that reasoning about durative actions is reduced to that about simple actions. Furthermore, time is treated as a numerical property. The benefit is that classical non-temporal planners, with minor modifications (*e.g.* to handle non-deterministic increase of *now*), may be used for planning in temporal domains that are as expressive as the above-defined subset of PDDL can represent. In the next section, we use this observation and propose a simple encoding of temporal planning problems as constraint satisfaction problems.

Encoding Temporal Planning as a CSP

The idea of planning as a constraint satisfaction problem stems from (Kautz & Selman 1992), who encode planning into SAT. This is generalized by (Lopez & Bacchus 2003) by encoding propositional planning as a CSP and showing that their encoding subsumes GraphPlan (Blum & Furst 1995). Actually, Lopez and Bacchus implicitly utilize a declarative semantics of STRIPS, since they appeal to the precondition axioms and successor state axioms obtained from the problem description. Given the declarative semantics for the temporal subset of PDDL, extending their result with temporal features follows naturally.

The Variable Structure

To encode a temporal planning problem as a CSP, we impose a bound on the length of the plan (in terms of concurrent happenings). We start with the search for a plan of length $n = 1$. If a plan of length n is found, then we return it and stop; otherwise, we increment n by 1 and repeat, until either a plan is found, or n exceeds a limit in which case we declare that no plan exists and abort.

[8] S represents durative actions by their start and end happenings; A corresponding PDDL plan restores all happenings of durative actions $t_1 : a[t_2 - t_1]$ from *start*(a, t_1) and *end*(a, t_2) in S.

In search of a plan of length n, we create n variables $A_j^{(s)}$ for each ground[9] action A_j, where $0 \le s \le n-1$, and $n+1$ variables $\phi_i^{(s)}$ for each ground fluent ϕ_i, where $0 \le s \le n$.

Each action variable $A_j^{(s)}$ is binary, and denotes the happening of either a simple action or a start or end event of a durative action. $A_j^{(s)}$ is true if and only if the action instance A_j happens in the s^{th} set of concurrent actions.

We use two sorts of variables for fluent instances: Predicates are modeled with binary variables. They include the original ones in the domain like *Happy*, along with auxiliary ones used in the previous section for capturing the semantics of the temporal features, like *Performing*(*burnMatch*). Functions are modeled with integer variables,[10] to represent the numerical functions in the domain, like NW, as well as the auxiliary ones, like *now* and *since*(*makeWish*).

Figure 2 shows the conceptual structure of the variables in the search for a plan of length 6 to our running example. (Ignore for now the lines and value assignments.) Notice that we do not use an action's last argument to denote its happening time. Instead, the variable $\textit{now}^{(s+1)}$ is used directly to represent the happening time of the actions in the s^{th} step, as indicated by the thick gray arrow in Figure 2. (See also Rule 5 of the definition of (s^*) below.) In this way, we integrate Axioms (28) and (29) in the variable structure, so as to keep the number of ground actions finite despite the infinite number of possible happening times.

The Constraints

To ensure that all and only valid plans can be generated, we encode the axioms in the declarative semantics into constraints of a CSP. This involves mapping ground terms in the axioms to constraint variables defined above, which is achieved by the (s^*) operator defined inductively as

1. $(\neg\alpha)^{(s^*)} = \neg(\alpha)^{(s^*)}$;
2. $(\alpha \odot \beta)^{(s^*)} = (\alpha)^{(s^*)} \odot (\beta)^{(s^*)}$,
   where $\odot \in \{\wedge, \vee, \supset, \equiv, \text{arithmetic operators}\}$;
3. $(\mathcal{R}[c, \alpha])^{(s^*)} = (\alpha)^{\left((s+1)^*\right)}$;
4. $(\vec{\tau}(\vec{x}))^{(s^*)} = \text{TRUE}$, where $\vec{\tau}(\vec{x})$ are typing predicates;[11]
5. $t^{(s^*)} = (\textit{time}(c))^{(s^*)} = \textit{now}^{(s+1)}$;
6. $X^{(s^*)} = \bar{X}^{(s)}$, where X is a ground fluent other than t and *time*(c), and $\bar{X}$ its corresponding constraint variable;
7. $(A \in c)^{(s^*)} = \bar{A}^{(s)}$ and $(A \notin c)^{(s^*)} = \neg\bar{A}^{(s)}$, where A is a ground action and $\bar{A}$ its corresponding variable;
8. $x^{(s^*)} = x$, where x is a number or a free variable.

With this (s^*) operator, we define the following constraints.

[9] Grounding for both actions and fluents follows the typing specification. For action terms, we ignore the time parameter t; the representation of time will be made clear below.

[10] Some constraint solvers like Choco (Choco) also support real numbers.

[11] Typing is handled in the grounding phase, and thus does not appear as constraints.

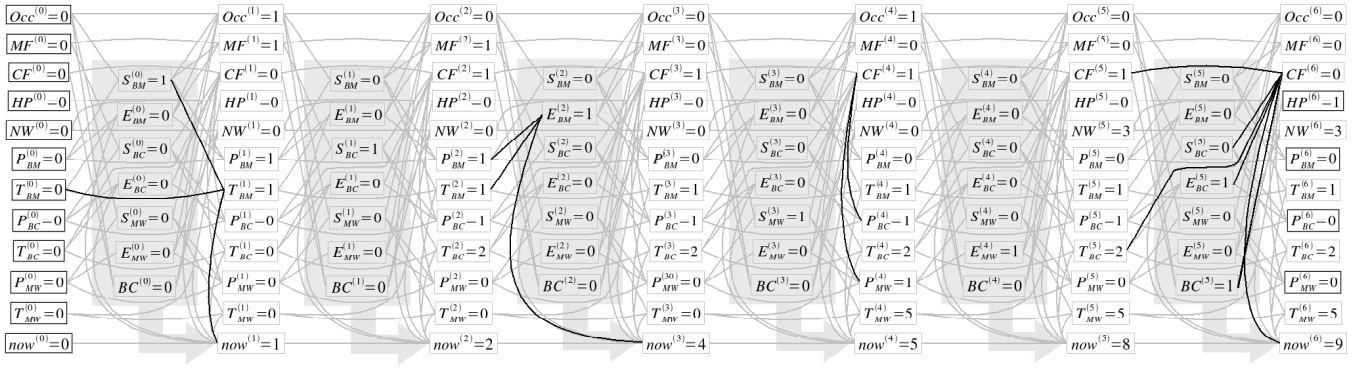

Figure 2: Conceptual variable structure and constraint network of the CSP encoding for the happy birthday-boy problem.
(Occ = *Occupied*, MF = *MatchFlame*, CF = *CandleFire*, HP = *Happy*, NW = *NumberOfWishes*, P_{BM} = *Performing*(*burnMatch*), P_{BC} = *Performing*(*burnCandle*), P_{MW} = *Performing*(*makeWish*), T_{BM} = *since*(*burnMatch*), T_{BC} = *since*(*burnCandle*), T_{MW} = *since*(*makeWish*), S_{BM} = *start*(*burnMatch*), E_{BM} = *end*(*burnMatch*), S_{BC} = *start*(*burnCandle*), E_{BC} = *end*(*burnCandle*), S_{MW} = *start*(*makeWish*), E_{MW} = *end*(*makeWish*) and BC = *blowCandle*.)

The initial state and goal constraints The fluent variables in the 0^{th} step represent the initial situation, hence must be set according to the initial database Σ_0 in the BAT. In particular, all variables for auxiliary fluents (*e.g.* $now^{(0)}$, $P_{BM}^{(0)}$, $T_{BC}^{(0)}$, *etc.*) are set to 0 (or FALSE).

The fluent variables in the n^{th} step describe the final situation. According to Theorem 1, for a valid plan, the goal formula must be satisfied in this situation and all *Performing* fluents must be FALSE. As an example, the goal constraint for our running example can be encoded as

$$Happy^{(6)} \wedge \neg P_{BM}^{(6)} \wedge \neg P_{BC}^{(6)} \wedge \neg P_{MW}^{(6)}$$

The action precondition constraint For each precondition axiom of the form $\Box Poss(A_j) \supset \pi_{A_j}$ in Σ_{pre}, including Axioms (9), (10), (18), (20) and (31), we assert in each concurrent step $0 \leq s \leq n-1$

$$(A_j \in c)^{(s^*)} \supset \pi_{A_j}^{(s^*)}$$

For example, the precondition for *end*(*burnMatch*) involves

$$\Box Poss(end(burnMatch, t)) \supset \\ Performing(burnMatch) \wedge t - since(A) = 3$$

so the corresponding constraint is

$$E_{BM}^{(s)} \supset P_{BM}^{(s)} \wedge (now^{(s+1)} - T_{BM}^{(s)} = 3)$$

for $s = \{0, \cdots, 5\}$, among which the constraint for $E_{BM}^{(2)}$ is highlighted by lines between related variables in Figure 2.

Invariant constraints and TIL enforcement

Notice that Axioms (25) and (33) are not handled as an precondition constraint, since the argument to $Poss(c)$ is a set. To encode axioms of the form

$$\Box Poss(c) \supset \Psi \tag{34}$$

we simply assert the constraint $\Psi^{(s^*)}$, since Ψ is required to hold for the concurrent step to be possible.

For example, the invariant axiom for *makeWish* requires

$$\Box Poss(c) \supset \mathcal{R}\big[c, \big(Performing(makeWish) \supset CF\big)\big]$$

so the corresponding constraint is $P_{MW}^{(s+1)} \supset CF^{(s+1)}$ for $s = \{0, \cdots, 5\}$ (Highlighted in Step 4 of Figure 2).

The successor state constraint For each SSA of fluent predicate of the form $\Box[c]F_i \equiv \Phi_{F_i}$, including Axioms (7), (15), (22), (23) and (32), we assert for $0 \leq s \leq n-1$

$$F_i^{\left((s+1)^*\right)} \equiv (\Phi_{F_i})^{(s^*)} \tag{35}$$

For example, the SSA for fluent CF is

$$\Box[c]CF \equiv \exists t.\, start(burnCandle, t) \in c \vee \\ CF \wedge \neg\exists t.\, \big(blowCandle(t) \in c \vee \\ end(burnCandle, t) \in c \wedge t - since(burnCandle) = 10\big)$$

which is encoded into the constraint

$$CF^{(s+1)} \equiv \big(S_{BC}^{(s)} \vee CF^{(s)} \wedge \\ \neg(E_{BC}^{(s)} \wedge (now^{(s+1)} - T_{BC}^{(s)} = 10) \vee BC^{(s)})\big)$$

for $s = \{0, \cdots, 5\}$ (Highlighted in Step 5 of Figure 2).

For each SSA of fluent functions $\Box[c]f_i = y \equiv \Phi_{f_i}$, including (8), (17) and (19), we assert for $0 \leq s \leq n-1$

$$\big((\Phi_{f_i})^{(s^*)}\big)_{f_i^{((s+1)^*)}}^{y}. \tag{36}$$

For example, the SSA for *since*(*burnMatch*) is

$$\Box[c]since(burnMatch) = y \equiv \\ \exists t.\, start(burnMatch, t) \in c \wedge y = time(c) \vee \\ since(burnMatch) = y \wedge \forall t.\, start(A, t) \notin c$$

so the corresponding successor state constraint is

$$S_{BM}^{(s)} \wedge (T_{BM}^{(s+1)} = now^{(s+1)}) \vee (T_{BM}^{(s)} = T_{BM}^{(s+1)}) \wedge \neg S_{BM}^{(s)}$$

for $s = \{0, \cdots, 5\}$ (Highlighted in Step 0 of Figure 2).

Non-null step constraints Axiom (27) ensures that no concurrent happening is an empty one. This can be encoded as[12] $\bigvee_j A_j^{(s)}$ for $0 \leq s \leq n-1$. These constraints help prune impossible branches when solving the CSP, since without them, a null step may exist, and the effective length of plan reduces to $n-1$, which has been proved impossible.

[12] Equivalent high-level constraints like "at-least-one" should be used for efficiency, whenever possible.

Problem	PDDL Plan
(a)	(1:A[4]), (3:B[2])
(b)	(1:A[4]), (4:B[2])
(c)	(1:A[4]), (4:B[2])
(d)	(1:A[5]), (3:B[4]), (4:C[1])
(e)	(1:burnMatch[3]), (2:burnCandle[7]), (5:makeWish[3]), (9:blowCandle)

Figure 3: Experimental results for problems in Figure 1

Mutex action constraints For each pair of actions A_{j_1} and A_{j_2} that is mutex according to the PDDL semantics, we add the constraint $\neg A_{j_1}^{(s)} \vee \neg A_{j_2}^{(s)}$. For example, $\neg E_{BM}^{(s)} \vee \neg S_{MW}^{(s)}$, since *end*(*burnMatch*) asserts $\neg$*Occ* while *Occ* is a start precondition of *makeWish*, violating the *no moving target* rule (Fox & Long 2003) for non-interference.

Action happening time constraint Finally, Axiom (30) ensures the chronological order of happenings. It is of the form in (34), so following the substitution rule there, it is encoded as $now^{(s)} < now^{(s+1)}$ for $s = 0, \cdots, n-1$. Notice that "$<$" requires that the next step happen strictly later than the current one, which ensures the *non-zero separation* condition (Fox & Long 2003).

The action happening time constraint introduced here, together with the duration constraints encoded in the precondition constraints, advances the time in the plan. To ensure that plans with short makespan are found, one needs to customize the search algorithm to try the domain values in increasing order, when solving the CSP.

Implementation and Test Results

We implement the encodings above in the Java-based constraint programming package Choco (Choco), due to at least two of its nice properties. First, it supports higher-order constraints, *i.e.* constraints over constraints. This makes it possible to express the complicated precondition and successor state constraints in the form of nested built-in constraints like `and`, `or`, `implies`, *etc.* This is distinct from (Lopez & Bacchus 2003), who encode them with non-standard multi*ary* constraints, which are far less efficient to propagate. Second, Choco supports real numbers, so numerical fluents may be modeled with reals in the future.

We run five test cases with the resulting CSP-based planner, which are exactly those shown in Figure 1. While most existing temporal planners are not powerful enough to handle them, our planner returns the correct plans within less than 1 second in all cases. The variable assignments in Figure 2 shows a sample solution to our running example, from which the PDDL plan can be extracted. Figure 3 shows the PDDL plans for all the five cases.

We have not yet empirically compared the efficiency with other state-of-the-art planners. However, run-time performance is less of a concern, since our main focus here is the temporally-expressive encoding and no optimization is ever taken into account at the current stage. Nevertheless, our encoding is a general one, and is orthogonal to many existing efficiency-improving techniques, so there are many ways to optimize the current base encoding.

Discussion

Contribution and Related Work

Along the line of temporal planning as CSP, Mali (2002) proposes a SAT encoding. However, it makes the TGP-like assumption (overall-condition end-effect), and is thus temporally simple. Moreover, it relies on the number of layers for representing time, so null-steps are allowed and the network becomes lengthy and inefficient when the durations of some actions are long. In contrast, our layers represent happenings, so no null-step exists, and the structure is much more compact. Besides, representing time as a numerical property and scheduling by solving constraints eliminates the drawback of decision epoch planners, since the choice of time is not predefined but instead arbitrary within the constraints. Finally, our encoding is derived from the formal declarative semantics, and is thus provably correct.

The constraint network, as suggested in Figure 2, appears similar to a planning graph (Blum & Furst 1995), but in fact there are a few differences. First, the connection between variables is a generalization of the edges in GraphPlan, since each layer in a planning graph is only influenced by the layer immediately before it, whereas in our representation, an action layer may have constraints with fact layers both before and after it, and a fact layer may have links with both the fact and action layers before it and the fact layer itself. In particular, effects are not represented as add and delete operations, but instead as successor state constraints of the form in (35) and (36). In this way, conditional effects can be concisely modeled without splitting an action into two. Second, numbers are accommodated in the network naturally due to the existence of numerical variables. This is crucial for our approach, since the happening times, represented by $now^{(s)}$, are modeled as a functional fluent using numerical variables. Finally, unlike in the planning graph, the constraints in different steps are the same, except for the initial and goal constraints. Therefore, we do not have the forward propagation and backward search phase. Instead, only a (uniform) constraint network construction and a CSP solution phase exist.

The idea of breaking durative actions into simple ones is not new. For example, Long and Fox (2003) propose LPGP, which splits a durative action into a start, an invariant and an end action, and uses a variant of GraphPlan to find a plan, whose happening times are then scheduled by a linear program solver. Unlike their approach, we only use two simple actions for each durative action, with the invariant condition protected by constraints derived from (25). This results in a more compact representation.

Due to the two-part search (GraphPlan plus LP scheduling), it is difficult for LPGP to handle duration-related effects, such as "the number of wishes is equal to the number of time units for making the wishes," since the duration of an action is still unknown in the plan generation phase, and thus the effect cannot be determined. Similar problems exist in VHPOP (Younes & Simmons 2003) and Tempo (Cushing

et al. 2007). Furthermore, LPGP and Tempo do not support duration inequalities. From this observation, it is interesting to notice that although LPGP, VHPOP and Tempo are temporally-expressive planners according to the definition of Cushing *et al.*, they are not PDDL-complete in that there are PDDL problems that they cannot solve.

In contrast, our approach unifies temporal planning and scheduling, and searches for actions and their happening times simultaneously in a single CSP. As a result, no matter how time, fluents and actions interact with one another, the search remains neutral and complete. This means that our planner is general enough to capture all of the problems that can be expressed within the target subset of PDDL.

Of course, there are "temporally expressive" planners outside PDDL, such as Zeno (Penberthy & Weld 1994), ASPEN (Fukunaga *et al.* 1997), IxTeT (Laborie & Ghallab 1995), *etc*. How they may generally handle PDDL problems, however, is yet to be further investigated.

Problems

One limitation of our approach is that the returned plan is optimal with respect to the number of concurrent steps, but not necessarily makespan optimal. One possible solution to this problem is to enforce a bound N on the length of plan, and let the planner find all plans of length $n \leq N$, and return the one with shortest makespan, as has been suggested in (Long & Fox 2003).

Ongoing and Future Research

We are implementing a general-purpose temporal planner based on the encoding introduced in this paper. We plan to investigate different optimization techniques to make it efficient and competitive.

We are also interested in applying our declarative semantics to state-space search planners, as well as finding more effective heuristics from the deeper insight that this brings.

Another possible research topic involves the definition of a declarative semantics, as well as the corresponding CSP encoding, for PDDL 3.0 (Gerevini & Long 2005) with constraints and preferences on the trajectory of plans.

Conclusion

In this paper, we introduced a new and general declarative semantics to the temporal subset of PDDL. Using it as the logical foundation, we defined a method to encode temporal planning problems directly into CSPs. This encoding captures most of the expressiveness of PDDL. Besides, our planner unifies plan generation and scheduling in one process. Both these properties contribute to the resulting planner being complete for temporal PDDL domains, which, to the best of our knowledge, no existing planner ever achieves.

Acknowledgment

The author would like to thank the anonymous reviewers, as well as Fahiem Bacchus, Jorge Baier, Jens Claßen and Christian Fritz for their comments. Special thanks to Hector Levesque for his revision suggestions and financial support.

References

Bacchus, F., and Ady, M. 2001. Planning with resources and concurrency: A forward chaining approach. In *IJCAI-01*.

Blum, A., and Furst, M. 1995. Fast planning through planning graph analysis. In *IJCAI-95*.

Chen, Y.; Wah, B.; and Hsu, C. 2006. Temporal planning using subgoal partitioning and resolution in SGPlan. *JAIR* 26:323–369.

Choco. http://choco.sourceforge.net.

Claßen, J.; Eyerich, P.; Lakemeyer, G.; and Nebel, B. 2007. Towards an integration of Golog and planning. In *IJCAI-07*.

Claßen, J.; Hu, Y.; and Lakemeyer, G. 2007. A situation-calculus semantics for an expressive fragment of PDDL. In *Proceedings of AAAI-07*.

Cushing, W.; Kambhampati, S.; Mausam; and Weld, D. S. 2007. When is temporal planning really temporal? In *IJCAI-07*.

Edelkamp, S., and Hoffmann, J. 2004. PDDL2.2: The language for the classical part of the 4th international planning competition. Tech. rep. 00195, Universität Freiburg.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *JAIR* 20:61–124.

Fukunaga, A.; Rabideau, G.; Chien, S.; and Yan, D. 1997. Towards an application framework for automated planning and scheduling. In *Proc. of IEEE Aerospace Conference*.

Gerevini, A., and Long, D. 2005. Plan constraints and preferences in PDDL3. Tech. rep., RT 2005-08-47, Dept. of Electronics for Automation, University of Brescia, Italy.

Hu, Y. 2006. A declarative semantics for a subset of PDDL with time and concurrency. Master's thesis, RWTH Aachen, Germany.

Kautz, H., and Selman, B. 1992. Planning as satisfiability. In *ECAI-92*.

Laborie, P., and Ghallab, M. 1995. Planning with sharable resource constraints. In *IJCAI-95*.

Lakemeyer, G., and Levesque, H. J. 2004. Situations, si! situation terms, no! In *KR-04*. AAAI Press.

Lifschitz, V. 1986. On the semantics of STRIPS. In Georgeff, M. P., and Lansky, A. L., eds., *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*.

Lin, F., and Reiter, R. 1997. How to progress a database. *Artificial Intelligence* 92:131–167.

Long, D., and Fox, M. 2003. Exploiting a GraphPlan framework in temporal planning. In *ICAPS-03*.

Lopez, A., and Bacchus, F. 2003. Generalizing GraphPlan by formulating planning as a CSP. In *IJCAI-03*.

Mali, A. D. 2002. On temporal planning as CSP. In *Proc. of IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, 75–82.

McDermott, D., and the AIPS'98 Planning Competition Committee. 1998. PDDL — the planning domain definition language. Tech. rep. www.cs.yale.edu/homes/dvm.

Penberthy, J. S., and Weld, D. S. 1994. Temporal planning with continuous change. In *Proceedings of AAAI-94*.

Reiter, R. 2001. *Knowledge in Action*. MIT Press.

Smith, D. E., and Weld, D. S. 1999. Temporal planning with mutual exclusion reasoning. In *IJCAI-99*.

Vidal, V., and Geffner, H. 2004. CPT: An optimal temporal POCL planner based on constraint programming. In *ICAPS-04*.

Younes, H. L., and Simmons, R. G. 2003. VHPOP: Versatile heuristic partial order planner. *JAIR* 20:405–430.