Adding Time and Intervals to Procedural and Hierarchical Control Specifications[*]

Tran Cao Son
Computer Science Department
New Mexico State University
Las Cruces, NM 88003, USA
tson@cs.nmsu.edu

Chitta Baral and Le-Chi Tuan
Computer Science and Engineering
Arizona State University
Tempe, AZ 85287, USA
chitta|lctuan@asu.edu

Abstract

In this paper we introduce the language Golog+HTN^{TI} for specifying control using procedural and HTN-based constructs together with deadlines and time restrictions. Our language starts with features from GOLOG and HTN and extends them so that we can deal with actions with duration by being able to specify time intervals between the start (or end) of an action (or a program) and the start (or end) of another action (or program). We then discuss an off-line interpreter based on the answer set planning paradigm such that the answer sets of the logic program have a one to one correspondence with the traces of the Golog+HTN^{TI} specification.

Introduction and Motivation

GOLOG (Levesque *et al.* 1997) is an Algol-like logic programming language for agent programming, control, and execution. It is based on the situation calculus theory of actions (Reiter 2000). GOLOG has been primarily used as a programming language for high-level agent control in dynamical environments (see e.g. (Burgard *et al.* 1998)). Some of the ways GOLOG can be extended are: (*a*) adding new program constructs (and providing the semantics for these constructs thereafter); or (*b*) adapting the language to an extension of situation calculus; or (*c*) combining (*a*) and (*b*). The first approach to extending GOLOG has been taken by (De Giacomo, Lespérance, & Levesque 2000; Baral & Son 1999) where various new constructs such as concurrency, interrupts, prioritized interrupts, or partial ordering are added to GOLOG. The other approaches can be seen in (Reiter 2001; Grosskreutz & Lakemeyer 2000; Boutilier *et al.* 2000) where continuous changes, durative actions, or stochastic actions are considered. In these extensions, time instances are introduced for specifying when an action (or a program) should be executed; in (Grosskreutz & Lakemeyer 2000), it is argued that the explicit specification of time in GOLOG programs – similar to what is used in (Reiter 2001) (also in (Boutilier *et al.* 2000)) – is not adequate for specifying event-driven behaviors and cc-Golog is proposed to address this problem. However, none of the constructs in cc-Golog allows us to express a simple behavior like "action a should start its execution within 3 units of time after action b starts its execution" because the language forces actions to happen as soon as possible, which – given that both a and b are executable at the time 0 – will not allow us to consider "a starts at 0 and b starts at 1" as an acceptable trajectory even though it satisfies the stated constraint.

In this paper, we investigate the introduction of actions with duration into the framework proposed in (Son, Baral, & McIlraith 2001) in which not only GOLOG-constructs such as sequence, while-loop, if-then-else, etc. but also HTN-based constructs such as the partial ordering and temporal constraints are allowed. This leads to a language[1], called Golog+HTN^{TI}, that generalizes GOLOG-based and HTN-based specifications with time intervals. Our language differs from the extensions of GOLOG in (Reiter 2001; Grosskreutz & Lakemeyer 2000; Boutilier *et al.* 2000) in that it includes different HTN-based constructs such as partial ordering and temporal constraints. Like cc-Golog, it does not require the explicit specification of time.

To characterize Golog+HTN^{TI} we need an action theory that allows actions to have durations. For that we chose a simple extension of the language $\mathcal{A}$ (Gelfond & Lifschitz 1998), which we will refer to as $\mathcal{AD}$. We give the semantics of $\mathcal{AD}$ using logic programming with answer sets semantics. This (the semantics of $\mathcal{AD}$) allows us to define the notion of a trajectory which we use to define the notion of a trace of a Golog+HTN^{TI} specification. We will begin with a short review of the answer set semantics of logic programs. We then present the language $\mathcal{AD}$ and discuss answer set planning with $\mathcal{AD}$ domains. We define the language Golog+HTN^{TI} and develop a logic programming based interpreter for it. Finally, we conclude with a brief discussion about future work.

Logic Programs and Answer Set Semantics

A logic program is a collection of rules of the form:

$$a_0 \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n \quad \text{or} \quad (1)$$

$$\leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n \quad (2)$$

where each of the a_i's is a literal in the sense of classical logic. not is the *negation-as-failure* connective. $not\ a$ is called a naf-literal. Intuitively, the first rule means that if $a_1, \ldots, a_m$ are true and $a_{m+1}, \ldots, a_n$ can be safely assumed to be false then a_0 must be true. The second rule is

[*]Supported by NSF grants EIA-0130887, EIA-0220590. NSF grant 0070463, and NASA grant NCC2-1232.

[1]The superscript TI refers to time and intervals.

a constraint that requires that at least one of $a_1, \ldots, a_m$ is false or one of $a_{m+1}, \ldots, a_n$ is true.

The body of a rule (1) or (2) is satisfied by a set of literals X if $\{a_1, \ldots, a_m\} \subseteq X$ and $\{a_{m+1}, \ldots, a_n\} \cap X = \emptyset$. A rule of the form (1) is satisfied by X if either its body is not satisfied by X or $a_0 \in X$. A rule of the form (2) is satisfied by X if its body is not satisfied by X.

For a set of literals X and a program P, the reduct of P with respect to X, denoted by P^X, is the program obtained from the set of all ground instances of P by deleting (*1.*) each rule that has a naf-literal $not\ l$ in its body with $l \in X$, and (*2.*) all naf-literals in the bodies of the remaining rules. Answer sets of logic programs are first defined by Gelfond and Lifschitz in (Gelfond & Lifschitz 1990). S is an *answer set* of P if it satisfies the following conditions.

1. If P does not contain any naf-literal (i.e. $m = n$ in every rule of P) then S is a minimal set of literals that satisfies all the rules in P.
2. If the program P does contain some naf-literal ($m < n$ in some rule of P), then S is an answer set of P if S is an answer set of P^S. (Note that P^S does not contain naf-literals, its answer set is defined in the first item.)

In what follows, we will refer to logic programming with answer set semantics as *AnsProlog*. Answer sets of propositional programs can be computed using answer set solvers such as **smodels** (Simons, Niemelä, & Soininen 2002), **dlv** (Eiter *et al.* 1998), **cmodels** (Lierler & Maratea 2003), or **ASSAT** (Lin & Zhao 2002). Answer set planning (ASP) (Subrahmanian & Zaniolo 1995; Lifschitz 2002) is an approach to planning using AnsProlog, an application of answer set programming (Marek & Truszczyński 1999; Niemelä 1999) to planning. In this approach, a planning problem is translated into a logic program whose answer sets correspond one-to-one to the solutions of the original problem. To make answer set programming easier, several new types of rules have been introduced. In this paper, we will often make use of *cardinality constraints* of the form: $l\{b_1, \ldots, b_k\}u \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n$ where a_i and b_j are literals and l and u are two integers, and $l \leq u$. Intuitively, such a rule enforces the constraint that if the body is true then at least l and at most u literals from the head are also true. Answer sets of programs with cardinality constraints are defined in (Simons, Niemelä, & Soininen 2002).

Reasoning About Durative Actions Using LP

As we mentioned earlier, to characterize domain constraints, we need to first describe an action description language. The action description language that we plan to use is a simple extension of the language $\mathcal{A}$ (Gelfond & Lifschitz 1998). In our extension, which we will refer to as $\mathcal{AD}$, we will allow actions to have duration and this will be sufficient to help us to justify and illustrate our language for domain constraints with new connectives.

Syntax of $\mathcal{AD}$

An action theory consists of two finite, disjoint sets of names **A** and **F**, called *actions* and *fluents*, respectively, and a set of propositions of the following form:

(3)	**causes**(a, f)	**initially**(f)	(5)
(4)	**executable**$(a, \{p_1, \ldots, p_n\})$	**duration**(a, v)	(6)

where f and p_i's are fluent literals (a *fluent literal* is either a fluent g or its negation $\neg g$) and a is an action. (3) is called a *dynamic causal law* and represents the effect of a while (4) states an executability condition of a. Intuitively, a proposition of the form (3) states that f is guaranteed to be true after the execution of a. An executability condition of a says that a is executable in a state in which $p_1, \ldots, p_n$ hold. Propositions of the form (5) are used to describe the initial state. It states that f holds in the initial state. Finally, a proposition of the form (6) is used to say that duration of action a is v.

An action theory is a set of propositions of the form (3)-(6). We will assume that each action a appears in one and only one proposition of the form (6) and v is a non-negative integer expression. We will often conveniently write $d(a)$ to denote the value v if **duration**$(a, v) \in D$ and for a set of actions A, $d(A) = \max\{d(a) \mid a \in A\}$.

Example 1 Consider an action theory with the set of fluents $\{f, g, h\}$, the set of actions $\{a, b, c, d\}$, and the following propositions:

causes(a, f)	**duration**$(a, 3)$	**executable**$(a, \{g, h\})$
causes(b, h)	**duration**$(b, 2)$	**executable**$(b, \{\})$
causes(c, g)	**duration**$(c, 2)$	**executable**$(c, \{\})$
causes$(d, \neg g)$	**duration**$(d, 1)$	**executable**$(d, \{\})$
initially$(\neg f)$	**initially**$(\neg g)$	**initially**$(\neg h)$

The propositions about a (the first three propositions on the first line) say that a will cause the fluent f to be true after 3 units of time and is executable only if g and h are true. The propositions for other actions have similar meaning.

Since the characterization of $\mathcal{AD}$ is not the aim of this paper, we do not present an independent characterization of it. (Recall that our goal is to use $\mathcal{AD}$ to show how to plan using a proposed domain constraint language Golog+HTN^{TI}.) Instead we give a AnsProlog encoding of prediction and planning using $\mathcal{AD}$. Moreover, a transition function based semantics for $\mathcal{AD}$ can be defined similarly to what has been done for the language $\mathcal{ADC}$ in (Baral, Son, & Tuan 2002). It is worth noticing that the definitions defined herein can be easily adapted to more complex action description language such as $\mathcal{ADC}$.

Semantics: Prediction in $\mathcal{AD}$

Given a set of propositions D we construct a logic program π_D for reasoning about the effects of actions in D. The main predicates in π_D are:

- $h(f, t)$ – fluent literal f holds at the time t;
- $exec(a, t)$ (resp. $in\_exec(a, t)$) – action a is executable (resp. in its execution) at t;
- $init(a, t)$ (resp. $ends(a, t)$) – action a starts (resp. ends) its execution at t;

The rules of π_D are given next.

- For each proposition (5) in D, π_D contains the following rule: $h(f, 1).$ (7)
  This describes the initial state (which fluents hold at time point 1) as specified by propositions of the from (5) in D.
- For each proposition (4) in D, π_D contains the rules:

$$\begin{array}{lcl} exec(a, T) & \leftarrow & not\ not\_exec(a, T). \\ not\_exec(a, T) & \leftarrow & not\ h(p_1, T). \\ & \ldots & \\ not\_exec(a, T) & \leftarrow & not\ h(p_n, T). \end{array} \qquad (8)$$

These rules define when a is executable at a time point T, based only on the truth of fluents. These rules establish that a is executable if its executability condition holds.

- For each proposition (6) in D, we add the following rules to π_D,

$$\begin{array}{lcl} ends(a, T+v) & \leftarrow & init(a,T). \\ in\_exec(a,T) & \leftarrow & init(a,T'), T' \leq T < T+v. \end{array} \quad (9)$$

These rules define when an action ends and when it is under execution.

- For each action a and an ef-proposition (3), the following rules are added to π_D,

$$\begin{array}{lcl} h(f,T) & \leftarrow & ends(a,T). \\ ab(f,T) & \leftarrow & in\_exec(a,T). \end{array} \quad (10)$$

These rules are used to reason about truth value of fluents at different time points.

Encoding the frame axiom. π_D contains the following rules that encode the frame axiom. They are slightly different from the normal logic program encoding of the frame axiom in (Gelfond & Lifschitz 1993). For each fluent f, the following rules belong to π_D:

$$\begin{array}{l} h(f, T+1) \leftarrow h(f,T),\ not\ ab(\neg f, T+1). \\ h(\neg f, T+1) \leftarrow h(\neg f, T),\ not\ ab(f, T+1). \end{array} \quad (11)$$

If we now want to find out if f would be true at time point t after starting the execution of actions a_1 at time point t_1, a_2 at time point t_2, ... and a_n at time t_n all we need to do is to add the set $\{init(a_i, t_i) \mid i \in \{t_1, \dots, t_n\}\}$ and the constraints $\leftarrow init(a_i, t_i), not\ exec(a_i, t_i)$ (for $i = 1, \dots, n$) to π_D, set the limits for the various variables, and ask if the resulting program entails $h(f, t)$.

One assumption we made in our characterization is that the effect of an action takes into account only after its execution ends, and the fluents, whose value changes due to an action execution, are in a unknown state during the execution. This of course can be changed by appropriately modifying π_D, in particular the rules in (10).

Answer Set Planning with $\mathcal{AD}$ Action Theories

We now show how the idea of answer set planning can be extended to $\mathcal{AD}$ action theories. Our AnsProlog planner for an action theory D, denoted by $\Pi(D)$, will consist of the program representing and reasoning about actions of D, π_D, the rules representing the goal, and the rules that generate action occurrences. Besides, we will need to set the limit on the maximal number of steps (the length) of the plan. We will call it $plan\_size$. From now on, whenever we refer to a time point t, we mean that $1 \leq t \leq plan\_size$.

Representing goal. Assume that we have a goal that is a conjunction of fluent literals $g_1 \wedge \dots \wedge g_m$. We represent this by a set of atoms $\{finally(g_i) \mid i = 1, \dots, m\}$. The following rules encode when the goal – as described by the $finally$ facts – is satisfied at a time point T.

$$\begin{array}{lcl} not\_goal(T) & \leftarrow & finally(X),\ not\ h(X,T). \\ goal(T) & \leftarrow & not\ not\_goal(T). \end{array} \quad (12)$$

The following constraint eliminates otherwise possible answer sets where the goal is not satisfied at the time point $plan\_size$.

$$\leftarrow\ not\ goal(plan\_size). \quad (13)$$

We now define the notion of a *plan*.

Definition 1 Given an action theory D, a goal G, and a plan size *plan_size*, we say that a sequence of sets of grounded actions $A_1, \dots, A_n$ is a *plan* achieving G if $goal(plan\_size)$ is true in every answer set of the program $\pi^{PVer}(D,G)$[2], which consists of

- the rules of π_D and the rules representing G (rules (7)-(13)) in which the time variable is less than or equal *plan_size*;
- the set of action occurrences $\bigcup_{i=1}^{n} \{init(a,i) \mid a \in A_i\}$;
- the rules preventing actions with contradictory conclusions to overlap (rules (15), below).

We say that a plan $p = A_1, \dots, A_n$ is a *concurrent plan* if there exists a pair i and j and an action $a \in A_i \cap A_j$ such that $i + d(a) > j$, i.e., p contains an overlapping of two instantiations of a same actions. p is said to be *non-concurrent* if it is not a concurrent plan.

Generating Action Occurrences. The following rules enumerate action initiations. To decrease the number of answer sets we have made the assumption that two action instantiations corresponding to the same action can not overlap each other, i.e., we consider only non-concurrent plans. This need not be the case in general. Our point here is that AnsProlog allows us to express such restrictions very easily. For each action a with the duration v, the rules:

$$\begin{array}{l} act(a,T) \leftarrow init(a,T_1), T_1 < T < T_1 + v. \\ ninit(a,T) \leftarrow not\ init(a,T). \\ init(a,T) \leftarrow exec(a,T), not\ act(a,T), not\ ninit(a,T). \end{array} \quad (14)$$

can be used to prevent two instantiations of a to overlap. The 1st rule defines when a is active and the 3rd rule allows a to occur only if it has not been initiated and is not active.

For every pair of a and b such that **causes**(a, f) and **causes**$(b, \neg f)$ belong to D, the two rules:

$$\begin{array}{l} overlap(a,b,T) \leftarrow in\_exec(a,T), in\_exec(b,T). \\ \leftarrow overlap(a,b,T). \end{array} \quad (15)$$

can be used to disallow actions with contradictory effects to overlap. Let $\pi^{PGen}(D,G)$ be the set of rules of $\Pi(D)$ with the goal G and $plan\_size{=}n$ and the rules (14)-(15). For an answer set M of $\pi^{PGen}(D,G)$, let $s_i(M) = \{f \mid h(f,i) \in M\}$ and $A_i(M) = \{a \mid init(a,i) \in M\}$. We can prove that there is an one-to-one correspondence between answer sets of $\pi^{PGen}(D,G)$ and non-concurrent plans achieving G.

Theorem 1 For a theory D and a goal G, $B_1, \dots, B_n$ is a non-concurrent plan that achieves G iff there exists an answer set M of $\pi^{PGen}(D,G)$ s.t. $A_i(M) {=} B_i$.

Golog+HTN^{TI}: Using Durations in Procedural and Hierarchical Domain Constraints

We begin with an informal discussion on the new construct in Golog+HTN^{TI}. Consider the domain from Example 1. It is easy to see that the program $b;c;a$ is a program achieving f from any state and the time needed to execute this plan is the sum of the actions's durations (Figure 1, Case (a)). Observe that b and c are two actions that achieve the condition for a to be executable and can be executed in parallel. Hence it should be obvious that any program that allows b and c to execute in parallel will have a shorter execution time. For the

[2]The ^{PVer} stands for *plan verification*.

moment, let us represent this by the program $p_1 = \{b, c\}; a$. The execution of this program is depicted in Figure 1, Case (b). Now consider a modification of the domain in Exam-

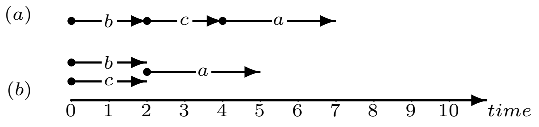

Figure 1: A pictorial view of program execution (the dot indicates when an action starts and the arrow indicates when an action stops)

ple 1, in which the executable propositions of c changes to **executable**$(c, \{\neg g\})$. A program achieving f would be to execute b and d in parallel, then c, and lastly a. We cannot execute b, c, and d in parallel all the time because c is not executable until $\neg g$ holds, and hence, it might need to wait for d to finish. It is easy to see, however, that it is better if c starts whenever d finishes. To account for this, we introduce a new construct that allows programs to start even if the preceding program has not finished. We write $(\{b, d\};^s_{[1,1]} c); a$ to indicate that c should start its execution 1 time unit after $\{b, d\}$ and then a and denote this program by p_2. The execution of this program can be illustrated as follows.

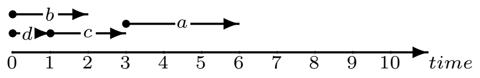

Figure 2: Execution of $(\{b, d\};^s_{[1,1]} c); a$

The above discussion provides a compelling argument for the need of *specifying time constraints in GOLOG programs and its extensions*. In the following, we extend the language proposed in (Son, Baral, & McIlraith 2001) with time constraints and define the language Golog+HTN^{TI}.

Definition 2 (Program) For an action theory D,

1. an action a is a program;
2. a temporal constraint $\phi[t_1, t_2]$, where ϕ is a fluent formula (a formula constructed using fluent literals and the propositional connectives), is a program;
3. if p_1 and p_2 are programs and $0 \leq t_1 \leq t_2$ are two time non-negative integers then so are $(p_1|p_2)$, $(p_1;^s_{[t_1,t_2]} p_2)$, and $(p_1;^e_{[t_1,t_2]} p_2)$;
4. if p_1 and p_2 are programs and ϕ is a fluent formula then so are "**if** ϕ **then** p_1 **else** p_2" and "**while** ϕ **do** p";
5. if X is a variable and $p(X)$ is a program then **pick**$(X, p(X))$ is a program[3];
6. if $p_1, \ldots, p_n$ are programs then a pair (S, C) is a program where $S = \{p_1, \ldots, p_n\}$ and C is a set of constraints over S of the following form: (i) $p_1 \prec^s_{[t_1,t_2]} p_2$ (or $p_1 \prec^e_{[t_1,t_2]} p_2$), (ii) $(p, \phi[t_1, t_2])$, (iii) $(\phi[t_1, t_2], p)$, and (iv) $(p_1, \phi[t_1, t_2], p_2)$ where p, p_1, p_2 are programs, ϕ is a fluent formula and $0 \leq t_1 \leq t_2$.

The constructs 1-5 in the above definitions are generalizations of constructs in GOLOG (Levesque *et al.* 1997) and the construct 6 is a generalization of hierarchical task networks (HTN) (Sacerdoti 1974). The main difference is that some of them are attached to a time interval and/or a directive '^{s}'/'^{e}' which are introduced for the specification of time constraints between program components. Intuitively, $\phi[t_1, t_2]$ says that if it is executed at the time moment t then the fluent formula ϕ must hold during the interval $[t+t_1, t+t_2]$. The program $(p_1;^s_{[t_1,t_2]} p_2)$ states that p_2 should start its execution at least t_1 and at most t_2 units of time after p_1 *starts* whereas $(p_1;^e_{[t_1,t_2]} p_2)$ forces p_2 to wait for t ($t_1 \leq t \leq t_2$) units of time after p_1 *finishes*. It is easy to see that $(p_1;^s_{[0,0]} p_2)$ requires that p_1 and p_2 be executed in parallel whereas $(p_1;^e_{[0,0]} p_2)$ requires that p_2 starts its execution at the time p_1 finishes. Note that $(p_1;^e_{[0,0]} p_2)$ corresponds to the original notation $p_1; p_2$.

The constraints in item 6 above are similar to truth constraints and ordering constraints over tasks in HTN. Intuitively, $p_1 \prec^s_{[t_1,t_2]} p_2$ (resp. $p_1 \prec^e_{[t_1,t_2]} p_2$) specifies the order and the time constraint for p_2 to start its execution. It states that if p_1 begins (resp. ends) its execution at a time moment t then p_2 must start its execution during the interval $[t + t_1, t + t_2]$. Similarly, $(p, \phi[t_1, t_2])$ (resp. $(\phi[t_1, t_2], p)$) means that ϕ must hold from t_1 to t_2 immediately after (resp. before) p's execution. $(p_1, \phi[t_1, t_2], p_2)$ states that p_1 must start before p_2 and ϕ must hold t_1 units of time after p_1 starts until t_2 units of time before p_2 starts.

We note that there is a subtle difference between the constructs $;^e_{[t_1,t_2]}$ (resp. $;^s_{[t_1,t_2]}$) and $\prec^e_{[t_1,t_2]}$ (resp. $\prec^s_{[t_1,t_2]}$) in that the former (inspired by GOLOG) represents a sequential order and the latter (inspired by HTN) represents a partial order between programs. For example, during the execution of the program $(p_1;^s_{[t_1,t_2]} p_2)$, no other program should start its execution. On the other hand, $(p_1 \prec^s_{[t_1,t_2]} p_2)$ only requires that p_2 should start its execution between the interval $[t_1, t_2]$ after p_1 starts its execution and does not prevent another program to start during the execution of p_1 and p_2.

Example 2 In our notation, p_1 and p_2 (from the discussion before Figure 1) are represented by $((b;^s_{[0,0]} c);^e_{[0,0]} a)$ and $(((b;^s_{[0,0]} d);^s_{[1,1]} c);^e_{[0,0]} a)$, respectively.

We will now define the notion of *a trace of a program*, which describes what actions are done when. But first we need to define the notion of a trajectory. For an action theory D and an integer n, let $\pi^{Gen}(D)$ be the set of rules (7)-(11) and (14)-(15) whose time variable belongs to $\{1, \ldots, n\}$.

Definition 3 (Trajectory) For an action theory D and an answer set M of $\pi^{PGen}(D)$, let $s_i = \{f \mid h(f, i) \in M\}$ and $A_i = \{a \mid init(a, i) \in M\}$. We say that the sequence $\alpha = s_1 A_1 \ldots s_n A_n$ is a *trajectory* of D.

Intuitively, a trajectory is an alternating sequence of states and action occurrences $s_1 A_1, \ldots, s_n A_n$, where s_i is a state at time point i and A_i is the set of actions that are supposed to have started at time point i. Observe that because of the assumption that during the execution of an action, value of fluents affected by the action is unknown, i.e., the states s_i might be incomplete. However, s_i will be complete if there

[3]Roughly speaking, programs are allowed to contain variables whose domains are pre-defined. A program with variables is considered as a shorthand for the set of its ground instantiations.

exists no action that is active at i. The notion of a trace of a program will be defined in the next four definitions.

Definition 4 (Trace–Primitive Cases) A trajectory $\alpha = s_1 A_1 \dots s_n A_n$ is a trace of a program p if

- $p{=}a$, $n{=}d(a)$ and $A_1{=}\{a\}$ and $A_i{=}\emptyset$ for $i > 1$; or
- $p{=}\phi[t_1, t_2]$, $n{=}t_2$, $A_i{=}\emptyset$ for every i, and ϕ holds in s_t for $t_1 \le t \le t_2$.

The next definition deals with programs that are constructed using GOLOG-constructs ((Levesque *et al.* 1997)).

Definition 5 (Trace–Programs with GOLOG-Constructs) A trajectory $\alpha = s_1 A_1 \dots s_n A_n$ is a trace of a program p if one of the following is satisfied.

- $p = p_1 \mid p_2$, α is a trace of p_1 or α is a trace of p_2,
- $p =$ **if** ϕ **then** p_1 **else** p_2, α is a trace of p_1 and ϕ holds in s_1 or α is a trace of p_2 and $\neg\phi$ holds in s_1,
- $p =$ **while** ϕ **do** p_1, $n = 1$ and $\neg\phi$ holds in s_1 or ϕ holds in s_1 and there exists some i such that $s_1 A_1 \dots A_i$ is a trace of p_1 and $s_{i+1} A_{i+1} \dots A_n$ is a trace of p, or
- $p =$ **pick**$(X, q(X))$, then there exists a constant x such that α is a trace of $q(x)$.

The trace of each program is defined based on its structure. We next deal with the new connectives $;^s_{[t_1,t_2]}$ and $;^e_{[t_1,t_2]}$.

Definition 6 (Trace–Parallel and Overlapping Programs) A trajectory $\alpha = s_1 A_1 \dots s_n A_n$ is a trace of a program p if

- $p{=}p_1 ;^s_{[t_1,t_2]} p_2$, there exists two numbers t_3 and t_4 such that $t_1 + 1 \le t_3 \le t_2 + 1$ and $t_4 \le n$ (because the index of the trace starts from 1) and either (i) there exists a trace $s_1 B_1 \dots s_{t_4} B_{t_4}$ of p_1 and a trace $s_{t_3} C_{t_3} \dots s_n C_n$ of p_2 such that $A_i = B_i \cup C_i$ for every i; or (ii) $t_3 \le t_4$ and there exists a trace $s_1 B_1 \dots s_n B_n$ of p_1 and a trace $s_{t_3} C_{t_3} \dots s_{t_4} C_{t_4}$ of p_2 such that $A_i = B_i \cup C_i$ (we write $B_j = \emptyset$ or $C_j = \emptyset$ for indexes that do not belong to the trace of p_1 or p_2); or
- $p{=}p_1 ;^e_{[t_1,t_2]} p_2$, there exists two numbers t_3 and t_4 such that $t_1{+}t_3 \le t_4 \le t_2{+}t_3$ and $t_4 \le n$ and $s_1 A_1 \dots s_{t_3} A_{t_3}$ is a trace of p_1 and a trace $s_{t_4} A_{t_4} \dots s_n A_n$ is a trace of p_2 and $A_i = \emptyset$ for every $t_3 \le i < t_4$.

This definition is best illustrated using a picture.

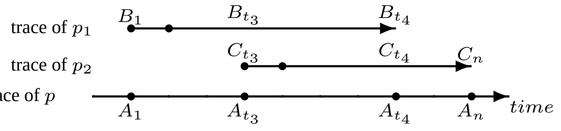

Figure 3: $A_i = B_i \cup C_i$ – First Item, Case 1 (Definition 6)

Example 3

- For $p_1{=}((b ;^s_{[0,0]} c) ;^e_{[0,0]} a)$ (wrt. the action theory in Example 1), we can easily check that $\mathbf{s^1_1}\ \{\mathbf{b}, \mathbf{c}\}\ \mathbf{s^1_2}\ \emptyset\ \mathbf{s^1_3} \{\mathbf{a}\}\ \mathbf{s^1_4}\ \emptyset\ \mathbf{s^1_5}\ \emptyset\ \mathbf{s^1_6}\ \emptyset$ where

  $s^1_1 = \{\neg f, \neg g, \neg h\}$ $\quad s^1_2 = \{\neg f\}$ $\quad s^1_3 = \{\neg f, g, h\}$
  $s^1_4 = \{g, h\}$ $\quad s^1_5 = \{g, h\}$ $\quad s^1_6 = \{f, g, h\}$

  is a trace of p_1. On the other hand, we can see that $\mathbf{s^1_1}\ \{\mathbf{b}, \mathbf{c}, \mathbf{d}\}\ \mathbf{s^1_2}\ \emptyset\ \mathbf{s^1_3} \{\mathbf{a}\}\ \mathbf{s^1_4}\ \emptyset\ \mathbf{s^1_5}\ \emptyset\ \mathbf{s^1_6}\ \emptyset$ is not a trace of p_1 although it contains a trace of p_1.
- For $p_2{=}(((b ;^s_{[0,0]} d) ;^s_{[1,1]} c) ;^e_{[0,0]} a)$ (wrt. the modified action theory), let

  $s^2_1 = \{\neg f, \neg g, \neg h\}$ $\quad s^2_2 = \{\neg f, \neg g\}$ $\quad s^2_3 = \{\neg f, h\}$
  $s^2_4 = \{\neg f, g, h\}$ $\quad s^2_5 = \{g, h\}$ $\quad s^2_6 = \{g, h\}$
  $s^2_7 = \{f, g, h\}$

  we can check that $\mathbf{s^2_1}\ \{\mathbf{b}, \mathbf{d}\}\ \mathbf{s^2_2}\ \{\mathbf{c}\}\ \mathbf{s^2_3}\ \emptyset\ \mathbf{s^2_4} \{\mathbf{a}\}\ \mathbf{s^2_5}\ \emptyset\ \mathbf{s^2_6}\ \emptyset\ \mathbf{s^2_7}\ \emptyset$ is a trace of p_2 but it is easy to see that $\mathbf{s^2_1}\ \{\mathbf{b}, \mathbf{d}\}\ \mathbf{s^2_2}\ \emptyset\ \mathbf{s^2_2}\ \{\mathbf{c}\}\ \mathbf{s^2_3}\ \emptyset\ \mathbf{s^2_4} \{\mathbf{a}\}\ \mathbf{s^2_5}\ \emptyset\ \mathbf{s^2_6}\ \emptyset\ \mathbf{s^2_7}\ \emptyset$ is not a trace of p_2 because c should start at the time moment 2.

We next deal with programs containing HTN-constructs.

Definition 7 (Trace–HTN Programs) A trajectory $\alpha{=}s_1 A_1 \dots s_n A_n$ is a trace of a program $p{=}(S, C)$ with $S{=}\{p_1, \dots, p_k\}$ if there exists two sequences of numbers $b_1, \dots, b_k$ and $e_1, \dots, e_k$ with $b_j \le e_j$, a permutation $(i_1, \dots, i_k)$ of $(1, \dots, k)$, and a sequence of traces $\alpha_j{=}s_{b_j} A^j_{b_j} \dots s_{e_j} A^j_{e_j}$ that satisfy the following conditions:

- for each l, $1 \le l \le k$, α_l is a trace of p_{i_l},
- if $p_t \prec p_l \in C$ then $i_t{<}i_l$,
- if $p_t {\prec^s_{[q_1,q_2]}} p_l {\in} C$ then $i_t{<}i_l$ and $b_{i_t}{+}q_1{\le}b_{i_l}{\le}b_{i_t}{+}q_2$,
- if $p_t {\prec^e_{[q_1,q_2]}} p_l {\in} C$ then $i_t{<}i_l$ and $e_{i_t}{+}q_1{\le}b_{i_l}{\le}e_{i_t}{+}q_2$,
- if $(\phi[t_1, t_2], p_l) \in C$ (or $(p_l, \phi[t_1, t_2]) \in C$) then ϕ holds in the states $s_{b_{i_l}-t_2}, \dots, s_{b_{i_l}-t_1}$ (or $s_{e_{i_l}+t_1}, \dots, s_{e_{i_l}+t_2}$), and
- if $(p_t, \phi[t_1, t_2], p_l) \in C$ then ϕ holds in $s_{b_{i_t}+t_1}$, …, $s_{b_{i_l}-t_2}$.
- $A_i = \cup^k_{j=1} A^j_i$ for every $i = 1, \dots, n$ where we assume that $A^j_i = \emptyset$ for $i{<}b_j$ or $i > e_j$.

The intuition of the above definition is as follows. First, each program starts (b_i's) and ends (e_i's) at some time point and it cannot finish before it even starts, hence, the requirement $b_i \le e_i$. The order of the execution is specified by the ordering constraints and not by the program's number. The permutation $(i_1, \dots, i_k)$ and j's record the starting time of the programs. The conditions on the trajectories make sure that the constraints are satisfied (first four items) and they indeed create $A_1, \dots, A_n$ (last item).

An AnsProlog Interpreter

We have developed an AnsProlog interpreter for programs defined in Definitions 2. For a program p of an action theory D, we define $\Pi(D, p)$ as a program consisting of $\Pi(D)$, the rules describing the program p, the set of rules for generating action occurrences (14)-(15), the constraint eliminating answer sets in which $trans(p, 1, plan\_size)$ does not hold, and a set of rules that realizes the operational semantics of programs (Definitions 4-7). We follow the approach in (Son, Baral, & McIlraith 2001) and define a predicate $trans(p, t_1, t_2)$ which holds in an answer set M of Π iff $s_{t_1}(M) A_{t_1}(M) \dots s_{t_2}(M) A_{t_2}(M)$ is a trace of p[4]. Space limitation does not allow a detailed presentation of Π. We therefore will concentrate on describing the ideas behind the rules and their meaning rather than presenting the rules in great detail. They can be found on our Web site[5].

We will begin with an informal discussion on the ideas behind the rules defining $trans(p, t_1, t_2)$. Intuitively, because of the rules (14)-(15), each answer set M of the program Π will contain a sequence of sets of actions $\alpha = A_1, \dots, A_n$ where $A_i = \{a \mid init(a, i) \in M\}$. The encoding of the action theory, π_D, makes sure that whenever an action a is

[4]For an answer set M, $s_i(M) = \{f \mid h(f, i) \in M,\ f$ is a fluent literal$\}$ and $A_i(M) = \{a \mid init(a, i) \in M\}$.

[5]URL: `http://www.cs.nmsu.edu/~tson/ASPlan/Duration`

initiated it is executable. Thus, the sequence α is a trajectory of D. So, it remains to be verified that α is indeed a trace of the program p. We will do this in two steps. First, we check if α contains a trace of p, i.e., we make sure that there is a trace $s_1 B_1 \ldots s_n B_n$ of p such that $B_i \subseteq A_i$. Second, we make sure that no action is initiated when it is not needed and define two predicates:

- $tr(p, t_1, t_2)$ - $s_{t_1} A_{t_1} \ldots A_{t_2}$ contains a trace of p;
- $used\_in(p, q, t, t_1, t_2)$ - a trace of p starting from t is used in constructing a trace of q from t_1 to t_2. Intuitively, this predicate records the actions belonging to the traces of q. The definition of this predicate will make sure that for a simple action a, only action a is used to construct its trace, i.e., $used\_in(a, a, t_1, t_1, t_1 + d(a))$ is equivalent to $init(a, t_1)$ and $used\_in(b, a, t_1, t_1, t_1 + d(a))$ is false for every $b \neq a$.

We define that $trans(p, t_1, t_2)$ holds iff $tr(p, t_1, t_2)$ holds and for every action $a {\in} A_j$ for $t_1 {\leq} j {\leq} t_2$, $used\_in(a, p, j, t_1, t_2)$ holds. The rules for $tr(p, t_1, t_2)$ are similar to the rules of the predicate $trans(p, t_1, t_2)$ from (Son, Baral, & McIlraith 2001) with changes that account for action duration and the new constructs such as $;^s_{[t_1,t_2]}$ and $;^e_{[t_1,t_2]}$ and checking for the condition of new constraint on a HTN-program. The construction of $\Pi(D, p)$ allows us to prove the following theorem.

Theorem 2 For a theory D and a program p, (i) for every answer set M of $\Pi(D, p)$, $s_1(M)A_1(M) \ldots s_n(M)A_n(M)$ is a trace of p, where $s_i(M) {=} \{f \mid h(f, i) {\in} M,\ f$ is a fluent literal$\}$ and $A_i(M) {=} \{a \mid init(a, i) {\in} M\}$; and (ii) if $s_1 B_1 s_2 \ldots s_n B_n$ is a trace of p then there exists an answer set M of $\Pi(D, p)$ such that $s_i = \{f \mid h(f, i) \in M\}$ and $B_i = \{a \mid init(a, i) \in M\}$.

Conclusion and Discussion

In this paper we propose a control specification language Golog+HTN^{TI} that generalizes procedural (based on GOLOG) and HTN-based specifications to allow time intervals. In the process we generalize the connective ';' to two connectives '$;^s_{[t_1,t_2]}$' and '$;^e_{[t_1,t_2]}$' and make similar generalizations of the HTN constructs. We then discuss the implementation of an AnsProlog-based interpreter for the language Golog+HTN^{TI}. Among the features that distinguish Golog+HTN^{TI} from previous extensions of GOLOG are:

1. The underlying action language of Golog+HTN^{TI} allows actions with duration and parallel execution;
2. The program constructs of Golog+HTN^{TI} generalize the constructs of GOLOG and HTN with time and intervals;
3. It does not require an explicit specification of time.

Finally, we notice that the approach presented in this paper can be extended to allow more complex action languages that allow continuous fluents and processes such as $\mathcal{ADC}$ in (Baral, Son, & Tuan 2002). Besides its use as a specification language similar to cc-Golog, Golog+HTN^{TI} can also be viewed as a language for specifying domain-dependent knowledge in a planner, a view explored by (Son, Baral, & McIlraith 2001) with respect to GOLOG. There, they demonstrated that a simplified version of programs discussed in this paper can help improving the performance of planner. To investigate this use of Golog+HTN^{TI}, we have developed a prototype of a planner for domains with continuous fluents and durative actions that uses Golog+HTN^{TI} with encouraging results. We will report our experiments with Golog+HTN^{TI} in planning in a future work.

References

Baral, C., and Son, T. C. 1999. Extending ConGolog to allow partial ordering. Theories, LNCS 1757, ATAL-99, 188–204.

Baral, C.; Son, T. C.; and Tuan, L.-C. 2002. A transition function based characterization of actions with delayed and continuous effects. KR-02, 291–302.

Boutilier, C.; Reiter, R.; Soutchanski, M.; and Thrun, S. 2000. Decision-theoretic, high-level agent programming in the situation calculus. AAAI-00, 355–362.

Burgard, W.; Cremers, A. B.; Fox, D.; Hähnel, D.; Lakemeyer, G.; D., S.; Steiner, W.; and Thrun, S. 1998. The interactive museum tour-guide robot. Artificial AAAI-98, 11–18.

Eiter, T.; Leone, N.; Mateis, C.; Pfeifer, G.; and Scarcello, F. 1998. The KR System `dlv`: Progress Report, Comparisons, and Benchmarks. KR-98, 406–417.

De Giacomo, G.; Lespérance, Y.; and Levesque, H. 2000. *ConGolog*, a concurrent programming language based on the situation calculus. AIJ 121(1-2):109–169.

Gelfond, M., and Lifschitz, V. 1990. Logic programs with classical negation, ICLP-90, 579–597.

Gelfond, M., and Lifschitz, V. 1993. Representing actions and change by logic programs. *JLP* 17(2,3,4):301–323.

Gelfond, M., and Lifschitz, V. 1998. Action languages. *ETAI* 3(6).

Grosskreutz, H., and Lakemeyer, G. 2000. cc-golog: Towards more realistic logic-based robot controllers. AAAI-00, 476–482.

Levesque, H.; Reiter, R.; Lesperance, Y.; Lin, F.; and Scherl, R. 1997. GOLOG: A logic programming language for dynamic domains. *JLP* 31(1-3):59–84.

Lierler, Y., and Maratea, M. Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. LPNMR-03, 346–350.

Lifschitz, V. 2002. Answer set programming and plan generation. *AIJ* 138(1–2):39–54.

Lin, F., and Zhao, Y. 2002. ASSAT: Computing Answer Sets of A Logic Program By SAT Solvers. AAAI-02.

Marek, V., and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. *The Logic Programming Paradigm: a 25-year Perspective*, 375–398.

Niemelä, I. 1999. Logic programming with stable model semantics as a constraint programming paradigm. *AMAI* 25, 241–273.

Reiter, R. 2000. On knowledge-based programming with sensing in the situation calculus. 2^{nd} *Int. Cog. Rob. Workshop, Berlin*.

Reiter, R. *KNOWLEDGE IN ACTION: Logical Foundations for Describing and Implementing Dynamical Systems*. MIT Press.

Sacerdoti, E. D. 1974. Planning in a hierarchy of abstraction spaces. *AIJ* 5:115–135.

Simons, P.; Niemelä, N.; and Soininen, T. 2002. Extending and Implementing the Stable Model Semantics. AIJ 138, 181–234.

Son, T. C.; Baral, C.; and McIlraith, S. 2001. Domain dependent knowledge in planning - an answer set planning approach. LPNMR-01, 226–239.

Subrahmanian, V., and Zaniolo, C. 1995. Relating stable models and ai planning domains. ICLP, 233–247.