Golog+HTN$^{TI}$: Adding time and intervals to procedural and hierarchical control knowledge

Chitta Baral  
Dept. of Computer Sc. & Eng.  
Arizona State University  
Tempe, AZ 85287, USA  
chitta@asu.edu

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

Le-Chi Tuan  
Dept. of Computer Sc. & Eng.  
Arizona State University  
Tempe, AZ 85287, USA  
lctuan@asu.edu

Abstract

In this paper we introduce a language for expressing procedural and HTN-based domain constraints. 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 of another action (or program). We then discuss a planner based on the answer set planning paradigm that can exploit such domain knowledge.

Introduction and Motivation

GOLOG (Levesque et al. 1997) is an Algol-like logic programming language for agent programming, and control and execution. It is based on a 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)). Although a planner can be written as a GOLOG program (See Chapter 10 of (Reiter 2000)), in (Son, Baral, & McIlraith 2001) an alternative view of GOLOG programs is presented. There it is viewed as an incompletely specified plan (or a form of procedural knowledge) that includes nondeterministic choice points that are filled in by the planner. For example, the GOLOG program $a_1; a_2; (a_3|a_4|a_5); f$ when viewed as a procedural knowledge tells a planner that the first action of the plan should be $a_1$, the second action should be $a_2$, and the third action should be one of $a_3$, $a_4$, and $a_5$ such that $f$ holds afterward. A planner, when given this procedural knowledge needs only to decide which one of $a_3$, $a_4$, or $a_5$ it should choose as its third action. In (Son, Baral, & McIlraith 2001) it is shown how such procedural knowledge can be exploited to speed up planning. There it is also shown how to combine constructs from HTN-planning with the GOLOG constructs.

Now suppose that actions have duration. In that case we may want to say that start $a_1$ and after 2 unit time of execution of $a_1$ start executing $a_2$ and so on. Similarly in an HTN-based domain constraint instead of the standard $a_1 < a_2$ – which means that $a_1$ should be executed before $a_2$ – we might want to say that $a_1$ should start at least 3 units of time before the start of $a_2$ and so on. To the best of our knowledge current extensions of GOLOG and HTN do not have such features. Thus our main goal in this paper is to develop a language that allows the expression of domain knowledge (procedural and HTN-based) of the above kind. We will refer to this language as Golog+HTN$^{TI}$, meaning that we add time and intervals to a language that has features from Golog and HTN.

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 $A$, which we refer to as $AD$. We give the semantics of $AD$ using logic programming with answer sets. This (the semantics of $AD$) allows us to define the notion of a trajectory. We use the notion of a trajectory to define the notion of a trace of a Golog+HTN$^{TI}$ specification. We also use logic programming with answer sets (by adding additional rules to the logic program that defines a trajectory) to give an alternative definition of trace and show the two notions to be equivalent. Finally we try to show that using Golog+HTN$^{TI}$ speeds up planning.

Background: LPASS and LP$^{smodels}$

We now briefly introduce LPASS and its extension LP$^{smodels}$. An LPASS program is a collection of rules of the form

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

where $a_i$'s are atoms. For an atom $a$, “not $a$” is referred to as a naf-literal. Intuitively, the above LPASS rule means that if $a_1 \ldots a_m$ are true and $a_{m+1} \ldots a_n$ can be assumed to be false then $a_0$ must be true.

The semantics of an LPASS program is defined using answer sets. LPASS programs whose rules do not have not in the body – referred to as definite programs – have unique answer sets, which are the least models of the theory obtained by treating rules of the form $a_0 \leftarrow a_1, \ldots, a_m$ as the classical formula $a_0 \land \ldots \land a_m \supseteq a_0$. Given an LPASS program $P$ and a set of atoms $S$, the Gelfond-Lifschitz transformation $P^S$ (Gelfond & Lifschitz 1990) is defined as the set of rules obtained from $P$ by removing all rules from $P$ whose body contains not $b$ such that $b \in S$, and then removing the naf-literals from the rest of the rules. A set $S$ of atoms is said to be an answer set of an LPASS program $P$ if $S$ is
the answer set of the definite program $P^S$. Answer sets of propositional LPASS programs can be computed using answer set solvers such as smodels (Niemelä & Simons 1997) and dlv (Citrinog et al. 1997). By $\text{LP}_{\text{smodels}}$, we refer to the extension of LPASS used in (Niemelä, Simons, & Soinnen 1999) where rules of the following form are also allowed:

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

where $a_i$ and $b_j$ are atoms 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 $u$ atoms from the head are also true.

### Reasoning about durative actions using LPASS

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 action description language that we plan to use is a simple extension of the language $A$ (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:

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

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. (2) is called a dynamic causal law and represents the effect of $a$ while (3) states an executability condition of $a$. Intuitively, a proposition of the form (2) 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 (4) are used to describe the initial state. It states that $f$ holds in the initial state. Finally, a proposition of the form (5) is used to say that duration of action $a$ is $v$.

An action theory is a set of propositions of the form (2)-(5). We will assume that each action $a$ appears in one and only one proposition of the form (5) and $v$ is a non-negative integer expression. We will often conveniently write $d(a)$ to denote the value $v$ if $\text{duration}(a, v) \in \mathcal{D}$ 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 $\{f, g, h\}$ and the set of actions $\{a, b, c, d\}$ and the following propositions:

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

The propositions about $a$ (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.

#### Encoding the frame axiom

In stead we give a LPASS encoding of prediction and planning using $\mathcal{AD}$. In another work, we present a transition function based characterization of $\mathcal{AD}$ and prove that it is equivalent to the LPASS characterization in this paper.

### Semantics: Prediction in $\mathcal{AD}$

Given a set of propositions $D$ we construct an LPASS program $\pi_D$ as follows:

1. For each $v$-proposition (4) in $D$, $\pi_D$ contains the following rule:
   \begin{equation}
   h(f, 1).
   \end{equation}
   This describes the initial state (which fluents hold in time point 1) as specified by the set of $v$-propositions in $D$.

2. For each $e$-proposition (3) in $D$, $\pi_D$ contains the following rules:
   \begin{align}
   \text{exec}(a, T) & \leftarrow \text{not not exec}(a, T). \tag{8} \\
   \text{not exec}(a, T) & \leftarrow \text{not} h(p, T). \tag{9} \\
   \text{...} \\
   \text{not exec}(a, T) & \leftarrow \text{not} h(p, T). \tag{10}
   \end{align}
   These rules define when $a$ is executable at a time point $T$, based only on the truth of fluents. Rules (9)-(10) say that $a$ is not executable if any of the fluent literals in its executability conditions does not hold.

3. For each $d$-proposition (5) in $D$, we add the following rules to $\pi_D$,
   \begin{align}
   \text{ends}(a, T + v) & \leftarrow \text{init}(a, T). \tag{11} \\
   \text{in exec}(a, T) & \leftarrow \text{init}(a, T'), T' < T < T + v. \tag{12}
   \end{align}
   These rules define when an action ends and when it is under execution.

4. For each action $a$ and an ef-proposition (2), the following rules are added to $\pi_D$,
   \begin{align}
   h(f, T) & \leftarrow \text{ends}(a, T). \tag{13} \\
   ab(f, T + 1) & \leftarrow \text{init}(a, T). \tag{14}
   \end{align}
   These rules are used to reason about truth value of fluents at different time points.

### Encoding the frame axiom

$\pi_D$ contains the following rules that encode the frame axiom. They are slightly different from the normal LPASS encoding of the frame axiom so as to take into account action duration.

\begin{align}
\text{h}(F, T + 1) & \leftarrow \text{h}(F, T), \text{not ab}(\neg F, T + 1). \tag{15} \\
\text{h}(\neg F, T + 1) & \leftarrow \text{h}(\neg F, T), \text{not ab}(F, T + 1). \tag{16}
\end{align}

If we now want to find out if $f$ would be true at time point $t$ after 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

$$\{\text{init}(a_i, t_i) \mid i \in \{t_1, \ldots, t_n\}\}$$

and the constraints

$$\text{not exec}(a_i, t_i)$$
(for \(i = 1, \ldots, n\)) to \(\pi_D\), set the limits for the various variables, and ask if the resulting program entails \(b(f, t)\).

One assumption we made in our characterization is that we assume that the effect of an action takes into effect only after its execution ends, and the fluents, whose value changes due to an action execution, are in an unknown state during the execution. This of course can be changed by appropriately modifying \(\pi_D\), in particular the rule (14).

**Answer Set Planning with AD Action Theories**

We now show how the idea of answer set planning (Lifschitz 1999) can be extended to AD action theories. Our LPASS planner for an action theory \(D\), denoted by \(\Pi(D)\), will consist of the program representing and reasoning about actions of \(D\), \(\pi_D\), 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 \(\text{plan\_size}\). From now on, whenever we refer to a time point \(t\), we mean that \(1 \leq t \leq \text{plan\_size}\).

**Representing goal.** Assume that we have a goal that is a conjunction of fluent literals \(q_1 \land \ldots \land q_m\). We represent this by a set of atoms \(\{\text{finally}(q_i) \mid i = 1, \ldots, m\}\). The following rules encode when the goal – as described by the finally facts – is satisfied at a time point \(T\):

\[
\begin{align*}
\text{not\_goal}(T) & \leftarrow \text{finally}(X), \text{not } h(X, T). \quad (17) \\
\text{goal}(T) & \leftarrow \text{not\_not\_goal}(T). \quad (18)
\end{align*}
\]

The following constraint eliminates otherwise possible answer sets where the goal is not satisfied at the time point \(\text{plan\_size}\).

\[\leftarrow \text{not\_goal}(\text{plan\_size}). \quad (19)\]

We now define the notion of a plan.

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

- the rules of \(\Pi(D)\) in which the time variable is less than or equal \(\text{plan\_size}\);
- the rules representing \(G\);
- the set of action occurrences
  \[
  \bigcup_{i=1}^{n} \{\text{init}(a, i) \mid a \in A_i\};
  \]
- the rules that disallow actions with contradictory conclusions to overlap (rules (23) and (24), below).

We say that a plan \(p = A_1, \ldots, A_n\) 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 LPASS allows us to express such restrictions very easily. For each action \(a\) with the duration \(v\) (i.e., \(\text{duration}(a, v)\) belongs to \(D\)), the following rule will be added to \(\Pi(D)\):

\[
\begin{align*}
\text{occc\_before}(a, T) & \leftarrow \text{init}(a, T_1), T_1 < T, T_1 + v. \quad (20) \\
\text{init}(a, T) & \leftarrow \text{exec}(a, T), \quad (21)
\end{align*}
\]

\[
\begin{align*}
\text{not\_occc\_before}(a, T) & \leftarrow \text{exec}(a, T), \quad (21) \\
\text{not\_init}(a, T) & \leftarrow \text{exec}(a, T). \quad (22)
\end{align*}
\]

The next rules disallow actions with contradictory effects to overlap: for every pair of \(a\) and \(b\) such that \(\text{causes}(a, f)\) and \(\text{causes}(b, f)\) belong to \(D\), \(\Pi(D)\) contains the two rules:

\[
\begin{align*}
\text{overlap}(a, b, T) & \leftarrow \text{in\_exec}(a, T), \text{in\_exec}(b, T). \quad (23) \\
\text{overlap}(a, b, T) & \leftarrow \text{in\_exec}(a, T), \text{in\_exec}(b, T). \quad (24)
\end{align*}
\]

Let \(\pi^{\text{PGen}}(D, G)\) be the set of rules of \(\Pi(D)\) with the goal \(G\) and \(\text{plan\_size}=n\). For an answer set \(M\) of \(\pi^{\text{PGen}}(D, G)\), let \(s_1(M) = \{f \mid h(f, i) \in M\}\) and \(A_i(M) = \{a \mid \text{init}(a, i) \in M\}\). We can prove that

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

**Golog+HTN T1: Using durations in Procedural and Hierarchical domain Constraints**

We begin with an informal discussion on the new construct in Golog+HTN T1. Consider the domain from Example 1. It is easy to see that the program \(b, c; a\) is a plan achieving \(f\) from any state and the time needed to execute this plan is the sum of the actions’ 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 plan that allows \(b\) and \(c\) to execute in parallel will have a shorter execution time. For the 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).

\[a \quad b \quad c \quad b \quad a \]

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

Now consider a modification of the domain in Example 1, in which the executable propositions of \(c\) changes to \(\text{executable}(c, \{\neg g\})\). 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 \(-g\) 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 \(((b, d); t_{[1,1]} c); a\) 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); \(t_{[1,1]} c\); a)](image)

We now define programs in Golog+HTN\(T^I\) that express domain knowledge to be exploited by a planner.

**Definition 2 (Program)** For an action theory \(D\),

1. an action \(a\) is a program;
2. a temporal constraint \(\phi[t_1, t_2]\) 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 \((p_1|p_2)\), \((p_1; t_{[1,2]} p_2)\), and \((p_1; t_{[1,2]} p_2)\);
4. if \(p_1\) and \(p_2\) are programs and \(\phi\) is a fluent formula then so are \(\text{if } \phi \text{ then } p_1 \text{ else } p_2\) and \(\text{while } \phi \text{ do } p\);
5. if \(X_1, \ldots, X_n\) are variables of sort \(s_1, \ldots, s_n\), respectively, \(p(X_1, \ldots, X_n)\) is a program, and \((f(X_1, \ldots, X_n))\) is a formula, then \(\text{pick}(X, f(X), p(X))\) is a program where \(X\) stands for \(X_1, \ldots, X_n\);
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) \(p_1 \prec \phi[t_1, t_2] p_2\) (or \(p_1 \prec t_{[1,2]} p_2\)),
ii) \((p, \phi[t_1, t_2])\),
iii) \((p_1, \phi[t_1, t_2], p),\)
iv) \((p_1, \phi[t_1, t_2], p_2)\)
where \(p, p_1, p_2\) are programs, \(\phi\) 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 GOLGY (Levesque et al. 1997) and the construct 6 is a generalization of hierarchical task networks (HTN) (Erol, Nau, & Subrahmanian 1995).

Intuitively, \(\phi[t_1, t_2]\) expresses the constraint that \(\phi\) holds during the time interval \([t_1, t_2]\) (from the current moment) and \((p_1; t_{[1,2]} p_2)\) 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 \((p_1|t_{[1,2]} p_2)\) forces \(p_2\) to wait for \(t_1 \leq t \leq t_2\) units of time after \(p_1\) finishes. It is easy to see that \((p_1; t_{[0,0]} p_2)\) requires that \(p_1\) and \(p_2\) be executed in parallel whereas \((p_1; t_{[0,0]} p_2)\) requires that \(p_2\) starts its execution at the time \(p_1\) finishes. Note that \((p_1; t_{[0,0]} p_2)\) 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, \(p_1 \prec \phi[t_1, t_2]\) says that \(p_2\) can start its execution after \(p_1\) is in execution (or after \(p_1\) finishes its execution) for \(t_1\) time units but no later than \(t_2\) time units of time. Similarly, \((p, \phi[t_1, t_2])\) means that \(\phi\) 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 \(\phi\) must hold \(t_1\) units of time after \(p_1\) starts until \(t_2\) units of time before \(p_2\) starts.

**Example 2** In our notation, \(p_1\) and \(p_2\) (from the discussion before Figure 1) are represented by \(((b; t_{0,0}) c); t_{[1,1]} c)\) and \(((b; t_{0,0}) d); t_{[1,1]} c)\) 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 \(P_{Gen}(D)\) be the program consisting of the set of rules of \(\Pi(D)\) whose time variable belongs to the set \(1, \ldots, n\) (recall that \(\Pi(D)\) consists of the rules (7)-(16) and (20)-(24)).

**Definition 3 (Trajectory)** For an action theory \(D\) and an answer set \(M\) of \(P_{Gen}(D)\), 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 occurred at time point \(i\). We are now ready to define what is a trace of a program. Since the definition of a trace is somewhat complicated we divide it into several small definitions and illustrate the complex ones with examples. First, we begin with the primitive cases.

**Definition 4 (Trace - Primitive Cases)** A trajectory \(\alpha = s_1 A_1 \ldots s_n A_n\) 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 \(\phi\) holds in \(s_i\) for \(t_1 = 2 \leq t_2\).

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

**Definition 5 (Trace - Programs with GOLOG-Construc-\(t\)\(ts\)** A trajectory \(\alpha = s_1 A_1 \ldots s_n A_n\) is a trace of a program \(p\) if one of the following is satisfied.

- \(p = p_1 | p_2, \alpha\) is a trace of \(p_1\) or \(\alpha\) is a trace of \(p_2\),
- \(p = \text{if } \phi \text{ then } p_1 \text{ else } p_2, \alpha\) is a trace of \(p_1\) and \(\phi\) holds in \(s_1\) or \(\alpha\) is a trace of \(p_2\) and \(\neg\phi\) holds in \(s_1\),
- \(p = \text{while } \phi \text{ do } p_1, n = 1\) and \(\neg\phi\) holds in \(s_1\) or \(\phi\) holds in \(s_1\) and there exists some \(i\) such that \(s_1 A_1 \ldots A_i\) is a trace of \(p_1\) and \(s_{i+1} A_{i+1} \ldots A_n\) is a trace of \(p\), or
- \(p = \text{pick}(X, f(X), q(X)), \) then there exists a constant \(\vec{x}\) of the sort of \(X\) such that \(f(\vec{x})\) holds in \(s_1\) and \(\alpha\) is a trace of \(q(\vec{x})\).

The trace of each program is defined based on its structure, i.e., how it is built. We next deal with the new connectives \(\phi[t_1, t_2]\) and \(\phi[t_{[1,2]}]\).

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

- \(p = p_1; t_{[1,2]} p_2\), there exists two numbers \(t_3\) and \(t_4\) such that \(t_1 + 1 \leq t_3 \leq t_2 + 1\) and \(t_4 \leq n\) (because the index of the trace starts from \(1\)) and either (i) there exists a trace \(s_1 B_1 \ldots s_4 B_4\) of \(p_1\) and a trace \(s_{t_3} C_{t_3} \ldots s_n C_n\) of \(p_2\) such
that $A_i = B_i \cup C_i$ for every $i$; or (ii) $t_3 \leq t_4$ and there exists a trace $s_1 B_1 \ldots s_n B_n$ of $p_1$ and a trace $s_1 C_1 \ldots s_n C_n$ of $p_2$ such that $A_t = B_i \cup C_i$ (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

- $p = p_1;_t [t_1, t_2] p_2$, there exists two numbers $t_3$ and $t_4$ such that $t_1 + t_3 \leq t_4 \leq t_3 + t_4$ and $n$ and $s_1 A_1 \ldots s_n A_n$ is a trace of $p_1$ and a trace $s_i A_1 \ldots s_i A_n$ is a trace of $p_2$ and $A_i = \emptyset$ for every $i \leq t_4$.

This definition is best illustrated using a picture.

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

Example 3 • For $p_1 = ((b^c_{[0,0]} c);^c_{[0,0]} a)$ (wrt. the action theory in Example 1), we can easily check that

$s_1^2 \{b, c\} s_2^2 \emptyset s_3^2 \{a\} s_4^2 = \emptyset s_5^2 \emptyset s_6^2$ \emptyset

where

- $s_1^2 = \{-f, -g, -h\}$,
- $s_2^2 = \{-f\}$,
- $s_3^2 = \{-f, g, h\}$,
- $s_4^2 = \{g, h\}$,
- $s_5^2 = \{g, h\}$,
- $s_6^2 = \{f, g, h\}$,

is a trace of $p_1$. On the other hand,

$s_1^2 \{b, c, d\} s_2^2 \emptyset s_3^2 \{a\} s_4^2 \emptyset s_5^2 \emptyset s_6^2$ \emptyset

although it contains a trace of $p_1$.

- For $p_2 = (((b^c_{[0,0]} d);^c_{[1,1]} c);^c_{[0,0]} a)$ (wrt. the modified action theory), we can easily check that

$s_2^1 \{b, d\} s_3^1 \{c\} s_4^1 \emptyset s_5^1 \emptyset s_6^1 \emptyset s_7^1 \emptyset

where

- $s_2^1 = \{-f, -g, -h\}$,
- $s_3^1 = \{-f, -g\}$,
- $s_4^1 = \{f, -g, h\}$,
- $s_5^1 = \{f, g, h\}$,
- $s_6^1 = \{g, h\}$,
- $s_7^1 = \{f, g, h\}$,

is a trace of $p_2$, but

$s_2^1 \{b, d\} s_3^1 \emptyset s_4^1 \{c\} s_5^1 \emptyset s_6^1 \emptyset s_7^1 \emptyset

is not a trace of $p_2$ because $c$ should start at the time moment 2.

In the next definition we deal with HTN-programs.

Definition 7 (Trace - HTN Programs) A trajectory $\alpha = s_1 A_1 \ldots s_n A_n$ is a trace of a program $p = (S, C)$ with $S = \{p_1, \ldots, p_k\}$ if there exists two sequences of numbers $b_1, \ldots, b_k$ and $e_1, \ldots, e_k$ with $b_j \leq c_j$, a permutation $(i_1, \ldots, i_k)$ of $(1, \ldots, k)$, and a sequence of traces $\alpha_j = s_{b_j} A_{i_j}^1 \ldots s_{e_j} A_{i_j}^2$ that satisfy the following conditions:

- for each $l$, $1 \leq l \leq k$, $\alpha_l$ is a trace of $p_{i_l}$,
- if $p_l \prec p_i \in C$ then $i_l < i_i$,
- if $p_l \prec p_i \in C$ then $i_l < i_i$ and $b_{i_l} + q_l \leq b_{i_i} \leq b_{i_l} + q_{i_l}$,
- if $p_l \prec p_i \in C$ then $i_l < i_i$ and $c_{i_l} + q_l \leq c_{i_l} \leq c_{i_l} + q_{i_l}$,
- if $(p_{i_l}, (\phi[t_1, t_2]), p_l) \in C$ (or $(p_{i_l}, \phi[t_1, t_2]) \in C$) then $\phi$ holds in the state $s_{b_{i_l}} t_{l-1} \ldots, s_{b_{i_l}} t_{l} (or s_{c_{i_l}} + t_{l-1} \ldots, s_{c_{i_l}} + t_{l})$, and
- if $(p_{i_l}, (\phi[t_1, t_2]), p_l) \in C$ then $\phi$ holds in $s_{b_{i_l}} t_{l-1} \ldots, s_{b_{i_l}} t_{l}$.

$A_i = \bigcup_{j=1}^k A_i^j$ for every $i = 1, \ldots, n$ where we assume that $A_i^1 = \emptyset$ 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 $b_i \leq e_i$. The order of the execution is specified by the ordering constraints and not by the program's number. The permutation $(i_1, \ldots, i_k)$ 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 $A_1, \ldots, A_n$ (last item).

An LPASS interpreter

We now present an LPASS interpreter for programs. We will adopt the way to encode formulas and programs in (Son, Baral, & McIraith 2001) for use with answer set solvers. In short, each program $p$ (resp. formula $\phi$) will be associated with a unique name $n_p$ (resp. $n_{\phi}$) and will be replaced by a set of rules and facts, denoted by $r(n_p)$ (resp. $r(n_{\phi})$). The formal definition of $r(n_p)$ and $r(n_{\phi})$ can be found in the aforementioned paper. Here, we demonstrate it by an example.

Example 4 $p_1 = ((b^c_{[0,0]} d);^c_{[1,1]} c);^c_{[0,0]} a)$ is encoded by the two atoms $proc(p_1, p_1, a, start, 0, 0)$ and $proc(p_1, b, c, start, 0, 0)$.

$p_2 = (((b^c_{[0,0]} d);^c_{[1,1]} c);^c_{[0,0]} a)$ is encoded by the atoms $proc(p_2, p_2, a, end, 0, 0)$, $proc(p_2, p_2, c, start, 1, 1)$, and $proc(p_2, b, c, start, 0, 0)$.

In the above, a program is specified by the predicate $\text{proc}$ with 5 arguments: the name, the first sub-program (the head), the rest of the program (the tail), whether the tail should start relative to the start or to the end of the head (start/end), and the minimal or the maximal time the tail needs to wait. Programs constructed using other constructs are encoded similarly. For example, the program $p = \text{if} \ \phi \ \text{then} \ p_1 \ \text{else} \ p_2$ is encoded by the atom $i(f(p, n_{\phi}, p_1, p_2)$ and the set of atoms encoding $p_1$ and $p_2$, and $\phi[t_1, t_2]$ by $\text{formula}(n_{\phi}, t_1, t_2)$ etc.

For a program $p$ of an action theory $D$, we define a logic program $\Pi$ that consists of the rules encoding the domain, $\pi_p$, the rules describing the program $r(n_p)$, the set of rules for generating action occurrences (20)-(24), and a set of rules that realizes the operational semantics of programs. We follow the approach in (Son, Baral, & McIraith 2001)
and define a predicate \( trans(p,t_1,t_2) \) which holds in an answer set \( M \) iff \( s_t(A_t(M),A_{t-1}(M)) \) is a trace of \( p^2 \). We will concentrate on describing the ideas behind the rules and their meaning rather than presenting the rules in great detail. We will present only a few representative rules. Readers interested in the source code of the program can obtain it from our web site.\(^3\)

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 (20)-(24), each answer set \( M \) of the program II will contain a sequence of sets of actions \( \alpha = A_1, \ldots, A_n \) where \( A_i = \{ a \mid init(a,i) \in M \} \). The encoding of the action theory, \( \pi_D \), makes sure that whenever an action \( a \) is initiated it is executable. Thus, the sequence \( \alpha \) is a trajectory of \( D \). So, it remains to be verified that \( \alpha \) is indeed a trace of the program \( p \). We will do this in two steps. First, we check if \( \alpha \) contains a trace of \( p \), i.e., we make sure that there is a trace \( s_{t_1}A_{t_1}, \ldots, A_{t_2} \) of \( p \) such that \( B_1 \subseteq A_1 \). Second, we make sure that no action is initiated when it is not needed. To do so, we define two predicates:

- \( tr(p,t_1,t_2) \) - a trace of \( p \) starting from \( t_1 \) is used in constructing a trace of \( q \) from \( t_1 \) to \( t_2 \).
- \( used \_ in(p,q,t_1,t_2) \) - if \( p \) is used in constructing \( q \) from \( t_1 \) to \( t_2 \).

The encoding of the predicate \( trans(p,t_1,t_2) \) is a trace of \( p \) if \( \alpha \) is a trajectory of \( D \) and \( \alpha \) is a trace of \( p \). So, it remains to be verified that \( \alpha \) is indeed a trace of the program \( p \). We will do this in two steps. First, we check if \( \alpha \) contains a trace of \( p \), i.e., we make sure that there is a trace \( s_{t_1}A_{t_1}, \ldots, A_{t_2} \) of \( p \) such that \( B_1 \subseteq A_1 \). Second, we make sure that no action is initiated when it is not needed. To do so, we define two predicates:

- \( tr(p,t_1,t_2) \) - a trace of \( p \) starting from \( t_1 \) is used in constructing a trace of \( q \) from \( t_1 \) to \( t_2 \).
- \( used \_ in(p,q,t_1,t_2) \) - if \( p \) is used in constructing \( q \) from \( t_1 \) to \( t_2 \).

The first rule accounts for the fact that \( used \_ in(a,a,t,t,t+d(a)) \) is always true if \( init(a,t) \) is true. The second rule says that if \( p \) is used in a trace of \( q \) and \( q \) is a trace of \( p \) then \( p \) is also in a trace of \( q \). These two rules are not enough, however, because so far we have not specified when a program

---

\(^2\)For an answer set \( M \), \( s_t(A_t(M),A_{t-1}(M)) \) is a trace of \( p^2 \).

\(^3\)http://www.cs.nmsu.edu/~tson/duration.
$p$ is used in a trace of $q$. This is done easily by using the weight rules. For instance, instead of the rule (26), we use the following rule:

$$3(tr(P, T_1, T_2), used_in(P_1, P, T_1, T_1, T_2),$$

$$\text{proc(P, P_1, P_2, start, M_1, M_2)},$$

$$T_1 + M_1 \leq T_2 \leq T_1 + M_2, T_1 \leq T_4 \leq T_2,$$

$$tr(P_1, T_1, T_2), tr(P_2, T_3, T_2).$$

(26')

The intuition of the above rule is that whenever we used the rule to derive $tr(P, T_1, T_2)$ then we also record what sub-programs are used and starting from which moment (by requiring that $used_in(P_1, P, T_1, T_1, T_2)$ and $used_in(P_2, P_3, T_3, T_1, T_2)$ also true). Similarly modification needs to be done for other rules defining $tr$. We omit them here for space reason.

Having defined $tr$ and $used_in$, defining $trans$ is simple: for every program $P$, $trans(P, T_1, T_2)$ holds only if $tr(P, T_1, T_2)$ holds and every action $A$ initiated during $T_1$ and $T_2 - 1$ must be used to construct a trace of $P$ from $T_1$ to $T_2$. So, we have the following rules:

$$\text{not_min}(P, T_1, T_2) \leftarrow action(A), init(A, T),$$

$$T_1 \leq T_2,$$

$$\text{not used_in}(A, P, T_1, T_2).$$

$$\text{trans}(P, T_1, T_2) \leftarrow T_1 \leq T_2, tr(P, T_1, T_2),$$

$$\text{not not_min}(P, T_1, T_2).$$

We illustrate this definition using a simple example.

Example 5 Consider $p_1 = ((b, a)_{[0,0]} c; [0,0] a)$ from Example 4 and the trajectory $\{b, c\}, \emptyset, \{a\}, \emptyset, \emptyset$.

For $p_1 = ((b, a)_{[0,0]} c; [0,0] a)$ (wrt. the action theory in Example 1) from Example 4 and the trajectory (Example 3)

$$s_1 = \{b, c\} s_2 = \emptyset s_3 = \{a\} s_4 = \emptyset s_5 = \emptyset s_6 = \emptyset$$

with

- $s_1 = \{\neg f, \neg g, \neg h\}$,
- $s_2 = \{\neg f\}$,
- $s_3 = \{\neg f, g, h\}$,
- $s_4 = \{g, h\}$,
- $s_5 = \{g, h\}$,
- $s_6 = \{f, g, h\}$.

Clearly, $tr(b, 1, 3), tr(c, 1, 3), tr(a, 3, 6)$ hold (because of rule (25)). $used_in(b, b, 1, 1, 3), used_in(c, c, 1, 1, 3)$, and $used_in(a, a, 3, 3, 6)$ hold (because of rule (33)). Furthermore, we have $tr(p_1, 1, 3), used_in(b, p_1, 1, 1, 3)$ and $used_in(c, p_1, 1, 1, 3)$ hold (because of rule (26')). Similarly, $tr(p_1, 1, 6), used_in(b, p_1, 1, 1, 6), used_in(c, p_1, 1, 1, 6)$, and $used_in(a, p_1, 3, 1, 6)$ hold (because of (26') and (34)).

This implies that $trans(p_1, 1, 6)$ holds. It is worth noting that $trans(b, 1, 3)$ does not hold since $c$ is initiated at 1 but $used_in(c, b, 1, 1, 3)$ does not hold.

We next present a theorem that extends a result from (Son, Baral, & McIlraith 2001) discussing a property of the above described program. Let $p$ be a program and $D$ be an action theory. Let $\Pi_n$ be the program consisting of

- the set of rules of $\Pi(D)$,
- the rules defining $tr$, $used_in$, and $trans$, and
- the rule expressing our goal of finding a trace of $p$

$$\leftarrow \text{not trans}(p, 1, n)$$

where all the time variables in $\Pi_n$ are bounded by $n$. Extending a result from (Son, Baral, & McIlraith 2001), we have the following theorem.

Theorem 2 For an action theory $D$ and a program $p$, (i) for every answer set $M$ of $\Pi_n$, $s_1(M)A_1 \ldots s_n(M)A_n(M)$ is a trace of $p$; and (ii) if $s_1B_1s_2 \ldots s_nB_n$ is a trace of $p$ then there exists an answer set $M$ of $\Pi_n$ such that $s_i = \{f \mid h(f, i) \in M\}$ and $B_i = \{a \mid init(a, i) \in M\}$.

Experimental Evaluations

We have used Golog+HTN \textit{III} in planning for domains from the AIPS2002 competition. We concentrate on formulating and testing our language in complex domains in which actions have durations, which might depend on the concrete situation. For example, the duration of filling a tank depends on various factors: the current level of fuel in the tank, the rate of which the fuel flows, etc. Of the 13 planners competed in the AIPS2002 competition, only 5 have this capability (see (Long et al.)). As an example, our planner \textit{5} can solve all 20 Zenotravi Flying problems in which the first seven problems were solved in less than 1 second, the next six problems were solved in less than 2 second and the last seven problems were solved within 10 to 60 seconds.

Conclusion

In this paper we proposed a domain constraint language Golog+HTN\textit{III} that generalizes procedural (based on GOLOG) and HTN-based domain constraints to allow time intervals. In the process we generalize the connective ‘;’ to two connectives ‘\textit{;}_{[t_1,t_2]}’ and ‘\textit{;}_{[t_1,t_2]}’ and similar generalizations of HTN constraints. We then show how the answer set planning paradigm can be used to plan where actions have duration, and we have domain constraints expressed in the language Golog+HTN\textit{III}. In terms of related work (that we have not mentioned yet) Reiter in (Reiter 2001) discusses temporal GOLOG where time is a parameter of the actions. Two differences (as relevant to the focus of this paper) between his approach and ours is that we generalize the connective ‘;‘ rather than actions, and we consider HTN-based constructs not considered by Reiter. Other differences include his use of Situation calculus (and first-order logic) as opposed to our use of a propositional action theory, cc-Golog (Grosskretz & Lakemeyer 2000) extended Con-Golog to allow time to be added to the program but cc-Golog concentrates on accommodating even-driven behavior rather than for planning.

\footnote{We contacted authors of other planners to help us setting up their planners so that a fair comparison between our planner and theirs can be done. Unfortunately, we were not able to finish this task in a timely fashion and did not yet have the comparison ready for this paper. We plan to have this issue resolved as soon as possible.}
References


