# Programming Hierarchical Task Networks in the Situation Calculus

**Alfredo Gabaldon**
Department of Computer Science
University of Toronto
alfredo@cs.toronto.edu

## Introduction

Hierarchical Task Network (HTN) planning (Sacerdoti 1974) is an approach to planning where problem-specific knowledge is used to remedy the computational intractability of classical planning. This knowledge is in the form of task decomposition directives, i.e. the planner is given a set of *methods* that tell it how a high-level task can be decomposed into lower-level tasks. The HTN planning problem consists in computing a sequence of primitive tasks that corresponds to performing the initial set of high-level tasks.

Our purpose in this paper is 1) to give an account of HTN-planning as high-level programming in the situation calculus (McCarthy 1963) based languages Golog/ConGolog (Levesque *et al.* 1997; De Giacomo, Lesperance, & Levesque 2000) and 2) to illustrate our approach with a ConGolog encoding of a logistics domain HTN-planning problem. The Golog/ConGolog languages have been extended to deal with explicit time, sensing actions, exogenous events, execution monitoring, incomplete knowledge of the initial state, stochastic actions and others. Thus the range of problems that can be tackled with this approach is potentially much larger. As an example, we modified the logistics domain encoding to execute on-line and deal with run-time exogenous delivery requests.

## Preliminaries

### The Situation Calculus

The situation calculus (McCarthy 1963) is a logical language for axiomatizing dynamic worlds. Intuitively, it has three basic components: *actions*: responsible for all the changes in the world; *situations*: sequences of actions which represent possible histories of the world; and *fluents*: relations and functions which represent properties of the world and whose values change from situation to situation.

We will use the definition of the situation calculus and the axiomatization of situations as it appears in (Levesque, Pirri, & Reiter 1998; Reiter 2001). The language of the situation calculus includes function symbols for actions, for example, $loadTrk(obj, trk)$ could stand for the action of loading $obj$ into truck $trk$. It includes a special constant $S_0$ that denotes the *initial situation* and a function symbol $do(\alpha, s)$ that denotes the situation that results from doing action $\alpha$ in situation $s$. For example, the situation term

$$do(driveTrk(Trk_1, Loc_1, Loc_2), do(loadTrk(A, Trk_1), S_0))$$

denotes the history of the world consisting of the sequence of actions

$$[loadTrk(A, Trk_1), driveTrk(Trk_1, Loc_1, Loc_2)].$$

*Relational fluents* and *functional fluents* are relations and functions, resp., whose last argument is a situation. Examples of these are a relation $atTruck(Trk_1, Loc_1, S_0)$ meaning that $Trk_1$ is at $Loc_1$ in the initial situation, and function $temperature(Room_1, s)$ denoting the temperature value of $Room_1$ in situation $s$.

A situation calculus axiomatization of a domain includes[1]:

1. Action precondition axioms: For each action function $A(\vec{x})$ an axiom of the form $Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)$ where $\Pi_A(\vec{x}, s)$ is a formula with free variables among $\vec{x}, s$ and $s$ is its only situation term. These axioms characterize the (situation dependent) preconditions for the execution of primitive actions.
2. Successor state axioms: For each relational fluent $F(\vec{x}, s)$ an axiom of the form $F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s)$ where $\Phi_F(\vec{x}, a, s)$ has free variables among $\vec{x}, a, s$ and $s$ is its only situation term. Similar axioms are included for functional fluents. These axioms characterize the value of fluents after executing a primitive action $a$ in situation $s$. These axioms embody Reiter's solution to the frame problem for deterministic actions (Reiter 1991).
3. Unique names axioms for actions.
4. Axioms describing the initial situation: A finite set of sentences whose only situation term is $S_0$ and which describe what is initially true, before any actions have occurred.

**Example 1** Our main example through out this paper will be a logistics domain problem. There are objects that are to be moved between locations by truck or plane. Cities contain different locations some of which are airports. Primitive actions include loading/unloading an object onto a truck or

[1]Arguments in predicates and formulas starting with a lower-case letter denote variables. Free variables are implicitly universally quantified.

plane, driving a truck and flying a plane. The following is an axiomatization of this domain:
Action Precondition Axioms:

$$
\begin{aligned}
&Poss(loadTruck(o,tr),s) \equiv \\
&\qquad atTruck(tr,l,s) \wedge atObj(o,l,s). \\
&Poss(unloadTruck(o,tr),s) \equiv inTruck(o,tr,s). \\
&Poss(loadAirplane(o,p),s) \equiv \\
&\qquad atObj(o,l,s) \wedge atAirplane(p,l,s). \\
&Poss(unloadAirplane(o,p),s) \equiv inAirplane(o,p,s). \\
&Poss(driveTruck(tr,orig,dest),s) \equiv \\
&\qquad atTruck(tr,orig,s) \wedge inCity(orig,city) \wedge \\
&\qquad\qquad inCity(dest,city). \\
&Poss(fly(p,orig,dest),s) \equiv \\
&\qquad atAirplane(p,orig,s) \wedge airport(dest).
\end{aligned}
$$

Successor State Axioms:

$$
\begin{aligned}
&atObj(o,l,do(a,s)) \equiv \\
&\qquad a = unloadTruck(o,tr) \wedge atTruck(tr,l,s) \vee \\
&\qquad a = unloadAirplane(o,p) \wedge atAirplane(p,l,s) \vee \\
&\qquad atObj(o,l,s) \wedge a \neq loadTruck(o,tr) \wedge \\
&\qquad\qquad a \neq loadAirplane(o,p). \\
&atTruck(tr,l,do(a,s)) \equiv \\
&\qquad a = driveTruck(tr,o,l) \vee \\
&\qquad atTruck(tr,l,s) \wedge (a \neq driveTruck(tr,o,d) \vee d = l). \\
&atAirplane(p,apt,do(a,s)) \equiv \\
&\qquad a = fly(p,oapt,apt) \vee atAirplane(p,apt,s) \wedge \\
&\qquad\qquad (a \neq fly(p,oapt,dapt) \vee dapt = apt). \\
&inTruck(o,tr,do(a,s)) \equiv \\
&\qquad a = loadTruck(o,tr) \vee \\
&\qquad\qquad inTruck(o,tr,s) \wedge a \neq unloadTruck(o,tr). \\
&inAirplane(o,p,do(a,s)) \equiv \\
&\qquad a = loadAirplane(o,p) \vee \\
&\qquad\qquad inAirplane(o,p,s) \wedge a \neq unloadAirplane(o,p).
\end{aligned}
$$

Unique names axioms for actions:

$$loadTruck(o,tr) \neq unloadTruck(o,tr),$$

$$loadTruck(o,tr) \neq loadAirplane(o,p), \text{ etc.}$$

Initial situation:

$$
\begin{aligned}
&atAirplane(p,l,S_0) \equiv \\
&\qquad p = Plane_1 \wedge l = Loc_{5,1} \vee p = Plane_2 \wedge l = Loc_{2,1}. \\
&atTruck(t,l,S_0) \equiv \\
&\qquad t = Truck_{1,1} \wedge l = Loc_{1,1} \vee \\
&\qquad t = Truck_{2,1} \wedge l = Loc_{2,1} \vee \ldots \\
&airport(loc) \equiv \\
&\qquad loc = Loc_{1,1} \vee loc = Loc_{2,1} \vee \\
&\qquad loc = Loc_{3,1} \vee loc = Loc_{4,1} \vee loc = Loc_{5,1}. \\
&inCity(l,c) \equiv \\
&\qquad l = Loc_{1,1} \wedge c = City_1 \vee \\
&\qquad l = Loc_{2,1} \wedge c = City_2 \vee \ldots \\
&atObj(p,l,S_0) \equiv \\
&\qquad p = Package_1 \wedge l = Loc_{3,3} \vee \\
&\qquad p = Package_2 \wedge l = Loc_{3,1} \vee \ldots
\end{aligned}
$$

The above set of axioms forms a complete situation calculus primitive action theory for our logistics domain example.

## Golog and ConGolog

The situation calculus based programming languages Golog (Levesque *et al.* 1997) and ConGolog (De Giacomo, Lesperance, & Levesque 2000) allow us to define complex actions in terms of the actions in a primitive action theory. The constructs of Golog are the following:

- Test condition: $\phi?$. Test whether $\phi$ is true in the current situation.
- Sequence: $\delta_1; \delta_2$. Execute $\delta_1$ followed by $\delta_2$.
- Nondeterministic action choice: $\delta_1 | \delta_2$. Execute $\delta_1$ or $\delta_2$.
- Nondeterministic choice of arguments: $(\pi x)\delta$. Choose a value for $x$ and execute $\delta$ for that value.
- Nondeterministic iteration: $\delta^*$. Execute $\delta$ zero or more times.
- Procedure definitions: **proc** $P(\vec{x})$ $\delta$ **endProc**. $P(\vec{x})$ is the name of the procedure, $\vec{x}$ its parameters, and $\delta$ is the body.

ConGolog has the above constructs plus the following:

- synchronized conditional: **if** $\phi$ **then** $\delta_1$ **else** $\delta_2$.
- synchronized loop: **while** $\phi$ **do** $\delta$.
- concurrent execution: $\delta_1 \parallel \delta_2$.
- prioritized concurrency: $\delta_1 \rangle\rangle \delta_2$. Execute $\delta_1$ and $\delta_2$ concurrently but $\delta_2$ executes only when $\delta_1$ is blocked or done.
- concurrent iteration: $\delta^{\parallel}$. Execute $\delta$ zero or more times in parallel.
- Interrupt: $\phi \rightarrow \delta$. Execute $\delta$ whenever condition $\phi$ is true.

**Example 2** The following is a procedure definition for the logistics domain:

**proc** $moveObj(o,loc)$
  $(\pi\ oloc, ocity)$.
  **if** $atObj(o,oloc) \wedge inCity(oloc,ocity)$ **then**
    %% if obj. is to be moved within the same city
    **if** $inCity(loc,ocity)$ **then** $inCityDeliver(o,oloc,loc)$
     **else** %% else must go by air to destination city
      $(\pi\ dcity)$.
      **if** $inCity(loc,dcity) \wedge dcity \neq ocity$ **then**
        $(\pi\ oaprt, daprt)$.
        $(inCity(oaprt,ocity) \wedge inCity(daprt,dcity))?$ ;
        $inCityDeliver(o,oloc,oaprt)$ ;
        $airDeliver(o,oaprt,daprt)$ ;
        $inCityDeliver(o,daprt,loc)$
      **else** $False?$
   **else** $False?$
**endProc**

The formal semantics of ConGolog is defined in terms of relations $Trans(\delta,s,\delta',s')$ and $Final(\delta,s)$.[2] Intuitively, $Trans(\delta,s,\delta',s')$ holds if after executing a single step of program $\delta$ in situation $s$, $\delta'$ is what remains of $\delta$ to be executed and $s'$ is the resulting situation. $Final(\delta,s)$ means that $\delta$ can be considered in a terminating state in situation $s$.

[2]For the original, simpler semantics of Golog see (Levesque *et al.* 1997; Reiter 2001).

These are some of the axioms for $Trans$ and $Final$ from (De Giacomo, Lesperance, & Levesque 2000):

$$
\begin{array}{l}
Trans(nil, s, \delta', s') \equiv False. \\
Trans(a, s, \delta', s') \equiv \\
\quad Poss(a, s) \wedge \delta' = nil \wedge s' = do(a, s). \\
Trans(\phi?, s, \delta', s') \equiv \\
\quad \phi[s] \wedge \delta' = nil \wedge s' = s. \\
Trans(\delta_1; \delta_2, s, \delta', s') \equiv \\
\quad (\exists\gamma)\delta' = (\gamma; \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee \\
\quad Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta', s'). \\
Trans((\pi x)\delta, s, \delta', s') \equiv \\
\quad (\exists x)Trans(\delta, s, \delta', s'). \\
Trans(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, s, \delta', s') \equiv \\
\quad \phi[s] \wedge Trans(\delta_1, s, \delta', s') \vee \\
\quad \neg\phi[s] \wedge Trans(\delta_2, s, \delta', s') \\
Trans(\textbf{while } \phi \textbf{ do } \delta, s, \delta', s') \equiv \\
\quad (\exists\gamma).(\delta' = \gamma\ ;\ \textbf{while } \phi \textbf{ do } \delta) \wedge \\
\qquad\qquad \phi[s] \wedge Trans(\delta, s, \gamma, s'). \\
Trans(\delta_1 \parallel \delta_2, s, \delta', s') \equiv \\
\quad (\exists\gamma)[\delta' = (\gamma \parallel \delta_2) \wedge Trans(\delta_1, s, \gamma, s')] \vee \\
\quad (\exists\gamma)[\delta' = (\delta_1 \parallel \gamma) \wedge Trans(\delta_2, s, \gamma, s')].
\end{array}
$$

$$
\begin{array}{l}
Final(nil, s) \equiv True. \\
Final(a, s) \equiv False. \\
Final(\phi?, s) \equiv False. \\
Final(\delta_1; \delta_2, s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s). \\
Final((\pi x)\delta, s) \equiv (\exists x)Final(\delta, s). \\
Final(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, s) \equiv \\
\quad \phi[s] \wedge Final(\delta_1, s) \vee \neg\phi[s] \wedge Final(\delta_2, s). \\
Final(\textbf{while } \phi \textbf{ do } \delta, s) \equiv \neg\phi[s] \vee Final(\delta, s). \\
Final(\delta_1 \parallel \delta_2, s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s).
\end{array}
$$

An abbreviation $Do(\delta, s, s')$, meaning that executing $\delta$ in situation $s$ is possible and it legally terminates in situation $s'$, can be defined in terms of the transitive closure of $Trans$ and predicate $Final$:

$$Do(\delta, s, s') \stackrel{\text{def}}{=} (\exists\delta').Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s').$$

$Trans^*$ is defined by a second order situation calculus formula. For details see (De Giacomo, Lesperance, & Levesque 2000).

A Prolog interpreter for ConGolog can be obtained almost directly from these axioms and a primitive action theory (De Giacomo, Lesperance, & Levesque 2000).

## HTN Planning

In this section we briefly review HTN-planning. Our discussion is based on the definitions of HTN-planning from (Erol, Hendler, & Nau 1996). For the primitive tasks, however, we will use situation calculus notation, i.e. we use primitive actions instead of STRIPS-style *HTN operators*. Moreover, we use situations instead of states.

A *primitive task* is an action term $A(\vec{x})$. A *compound task* is a term of the form $tname(\vec{x})$. A *task network* is a pair $(T, \phi)$ where $T$ is a list of tasks and $\phi$ a boolean formula of constraints of the forms $(t \prec t')$, $(t, l)$, $(l, t)$, $(t, l, t')$, $(v = v')$ and $(v = c)$ where $t, t'$ are tasks from $T$, $l$ is a fluent literal, $v, v'$ are variables and $c$ is a constant. An HTN *method* is a pair $(h, d)$ where $h$ is a compound task and $d$ is a task network. Methods are the HTN construct for building complex tasks from primitive ones.

An HTN *planning problem* is a tuple $(d, s, D)$ where $d$ is a task network, $s$ is a situation, and $D$ is a *planning domain* consisting of a primitive action theory plus a set of methods. A *plan* is a sequence of ground primitive tasks.

Let $d$ be a primitive task network, $s$ be a situation, and $D$ a planning domain. A sequence of primitive tasks $\sigma$ is a *completion* of $d$ in $s$, denoted by $\sigma \in comp(d, s, D)$, if $\sigma$ is a total ordering of a ground instance of the primitive task network $d$ and is executable in $s$.

Let $d$ be a task network that contains a compound task $t$ and $m = (h, d')$ be a method such that $\theta$ is a most general unifier of $t$ and $h$. Define $reduce(d, t, m)$ to be the task network obtained from $d\theta$ by replacing $t\theta$ with $d'\theta$ and incorporating (see (Erol, Hendler, & Nau 1996) for details) the constraints in $d'$ with those in $d$. Define $red(d, s, D)$ as the set of all reductions of $d$ by methods of $D$.

A solution $sol(d, S_0, D)$ to a planning problem $(d, S_0, D)$ is the set of all plans that can be computed in a finite number of reduction steps:

$$
\begin{array}{l}
sol_1(d, S_0, D) = comp(d, S_0, D) \\
sol_{n+1}(d, S_0, D) = \\
\qquad sol_n(d, S_0, D) \cup \bigcup_{d' \in red(d, S_0, D)} sol_n(d', S_0, D) \\
sol(d, S_0, D) = \bigcup_{n<\omega} sol_n(d, S_0, D)
\end{array}
$$

**Example 3** The following are methods for moving an object in the logistics domain that correspond to the Golog procedure example above. The first method works for moving an object within the same city. The second is for moving an object between cities.

$$
\begin{array}{l}
(moveObj(o, loc) \\
\quad [t = inCityDeliver(o, oloc, loc)] \\
\qquad (atObj(o, oloc), t) \wedge \\
\qquad (inCity(oloc, ocity), t) \wedge \\
\qquad (inCity(loc, ocity), t) \\
)
\end{array}
$$

$$
\begin{array}{l}
(moveObj(o, loc) \\
\quad [t_1 = inCityDeliver(o, oloc, oaprt), \\
\quad t_2 = airDeliver(o, oaprt, daprt), \\
\quad t_3 = inCityDeliver(o, daprt, loc)] \\
\qquad (atObj(o, oloc), t_1) \wedge (inCity(oloc, ocity), t_1) \wedge \\
\qquad (inCity(loc, dcity), t_1) \wedge (ocity \neq dcity, t_1) \wedge \\
\qquad (inCity(oaprt, ocity), t_1) \wedge \\
\qquad (inCity(daprt, dcity), t_1) \wedge \\
\qquad (t_1 \prec t_2) \wedge (t_2 \prec t_3) \\
)
\end{array}
$$

## Programming HTNs in Golog/ConGolog

In this section we show how HTN-planning problems can be encoded in Golog/ConGolog. Let us first consider task networks which are totally ordered and with a constraint formula $\phi$ that is a conjunction of constraints of the form $(l, t)$. This is the type of task networks the HTN-planning system SHOP (Nau *et al.* 1999) is designed to solve.

Totally ordered task networks can be encoded in Golog since there is no concurrency among the tasks.

## Totally ordered task networks

Consider an HTN-planning problem $P = (d, S_0, D)$. We encode the methods $(h, d_1), (h, d_2), \ldots, (h, d_k)$ of each compound task $h$ as a Golog procedure as follows:

$$\begin{array}{l} \textbf{proc}\ h \\ (L_{1,1})?\ ;\ t_{1,1}\ ;\ldots;(L_{1,i_1})?\ ;\ t_{1,i_1}\ | \\ (L_{2,1})?\ ;\ t_{2,1}\ ;\ldots;(L_{2,i_2})?\ ;\ t_{2,i_2}\ | \\ \ldots \\ (L_{k,1})?\ ;\ t_{k,1}\ ;\ldots;(L_{k,i_k})?\ ;\ t_{k,i_k} \\ \textbf{endProc} \end{array}$$

where $t_{i,j}$ is the $j$th task in $d_i$ and $L_{i,j}$ is a conjunction of the literals $l$ such that $(l, t_{i,j})$ is a constraint in $d_i$.

Let $\Delta_P$ denote the resulting set of Golog procedures. To complete the encoding of the HTN planning problem $P$ we include a Golog program $\delta_T$ obtained from the task network $d$. This program has the same form as that of a single method: $(L_1)?\ ;\ t_1\ ;\ldots;\ (L_l)?\ ;\ t_l$.

The HTN planning problem can now be reformulated in terms of the logical semantics of Golog:

$$\mathcal{D}_P \models (\exists s) Do(\Delta_P\ ;\ \delta_T, S_0, s)$$

Here, $\mathcal{D}_P$ is the primitive action theory of $P$ plus the axioms of Golog.

The procedure in Example 2 is an encoding of the methods in Example 3, except that instead of using nondeterministic choice of actions, i.e. operator $|$, we used $if$-statements since the conditions before the first tasks are mutually exclusive.

## Partially ordered task networks

Before we move on to partially ordered task networks, let us comment on enforcing constraints of the values of literals, i.e. constraints of the forms $(l, t)$, $(t, l)$ and $(t, l, t')$ and their boolean combination. Intuitively, one way to think about these constraints is that their purpose is for eliminating or "pruning" some of the plan candidates. Their purpose is similar to that of the temporal constraints used by Bacchus and Kabanza (1995; 2000) for controlling search in a forward chaining classical planner. Reiter uses this technique in a Golog implementation of several classical planners (Reiter 2001). The idea is to use a predicate $badSituation(s)$ to encode constraints and check them before adding a primitive action to the plan being computed. So in the remainder of the paper, we will assume that these constraints have been suitably encoded by means of a $badSituation$ predicate.

Furthermore, we will assume that the partial order boolean formula is a conjunction of atoms $(t \prec t')$. This is not a limitation since an unrestricted formula can also be enforced through the $badSituation$ predicate. However, if the partial order formula is a conjunction, it is computationally better to enforce it imperatively in the program.

Let us now consider encoding partial order HTN planning problems in ConGolog. As before, for each method there will be a procedure, but we also need to introduce two fluents and two actions which are used to enforce the partial ordering among tasks: fluent $executing(p(\vec{x}), s)$ meaning that the ConGolog procedure $p$ is executing in situation $s$, fluent $terminated(p(\vec{x}), s)$ meaning that the basic action or procedure $p$ has executed and terminated in situation $s$, action $start(p(\vec{x}))$ which causes $executing(p(\vec{x}), s)$ to become true, and $end(p(\vec{x}))$ which causes $executing(p(\vec{x}), s)$ to become false and $terminated(p(\vec{x}), s)$ to become true. Both fluents are initially false for all procedures and actions and the two actions are the only ones that change these fluents' truth value. Formally, the successor state axioms for these fluents are the following:

$$\begin{array}{l} executing(p(\vec{x}), do(a, s)) \equiv \\ \qquad a = start(p(\vec{x})) \lor \\ \qquad executing(p(\vec{x}), s) \land a \neq end(p(\vec{x})). \end{array}$$

$$\begin{array}{l} terminated(p(\vec{x}), do(a, s)) \equiv \\ \qquad a = p(\vec{x}) \lor a = end(p(\vec{x})) \lor \\ \qquad terminated(p(\vec{x}), s). \end{array}$$

Let $d$ be a task network and $t$ one of its tasks. Let $nexec(t)$ stand for $\neg executing(t) \land \neg terminated(t)$. Let $pred(t, d)$ stand for the conjunction:

$$\bigwedge_{\{t' : (t' \prec t) \in d\}} terminated(t)$$

If there is no constraint $(t \prec t_i)$ in $d$ then $pred(t, d) = True$.

The ConGolog procedure that encodes the methods $(h, d_1), (h, d_2), \ldots, (h, d_k)$ for a compound task $h$ is:

$$\textbf{proc}\ h\ \delta_1 | \delta_2 | \ldots | \delta_k\ \textbf{endProc}$$

where

$$\delta_i \stackrel{\text{def}}{=} \begin{array}{l} pred(t_{i,1}) \land nexec(t_{i,1}) \rightarrow t_{i,1}\ || \\ pred(t_{i,2}) \land nexec(t_{i,2}) \rightarrow t_{i,2}\ || \\ \ldots \\ pred(t_{i,k_i}) \land nexec(t_{i,k_i}) \rightarrow t_{i,k_i} \end{array}$$

The $t_{i,j}$s are the tasks in $d_i$. The $\delta_i$s consist of a set of interrupts one for each subtask. As soon as the predecessors of a task that has not yet executed terminate, the interrupt fires and the task executes.

**Example 4** This is a simple blocks world example method for moving a block $v_1$ from a block $v_2$ onto a block $v_3$:

$$\begin{array}{l} (move(v_1, v_2, v_3) \\ \quad [clear(v_1),\ clear(v_3),\ unstack(v_1, v_2), stack(v_1, v_3)] \\ \qquad (clear(v_1) \prec unstack(v_1, v_2)) \land \\ \qquad (clear(v_3) \prec unstack(v_1, v_2)) \land \\ \qquad (unstack(v_1, v_2) \prec stack(v_1, v_3)) \\ ) \end{array}$$

The encoding as a ConGolog procedure is the following:

$$\begin{array}{l} \textbf{proc}\ move(v_1, v_2, v_3) \\ \quad nexec(clear(v_1)) \rightarrow clear(v_1)\ || \\ \quad nexec(clear(v_3)) \rightarrow clear(v_3)\ || \\ \quad nexec(unstack(v_1, v_2))\ \land terminated(clear(v_1)) \land \\ \qquad\qquad terminated(clear(v_3)) \rightarrow unstack(v_1, v_3)\ || \\ \quad nexec(stack(v_1, v_3))\ \land terminated(unstack(v_1, v_2)) \\ \qquad\qquad \rightarrow stack(v_1, v_3) \\ \textbf{endProc} \end{array}$$

It is not always possible but in many cases the partial ordering of tasks can be captured without introducing extra fluents. For instance, the procedure for $move(v_1, v_2, v_3)$ can clearly be written in the following simpler way:

$$\begin{array}{l}\textbf{proc}\ move(v_1, v_2, v_3)\\ \quad (clear(v_1) \parallel clear(v_3))\ ;\\ \quad\quad unstack(v_1, v_2)\ ;\ stack(v_1, v_3)\\ \textbf{endProc}\end{array}$$

## On-line Execution with Exogenous Actions

The situation calculus and Golog/ConGolog are very powerful languages which allow one to solve problems well beyond the capabilities of today's HTN-planners. In this section we present an encoding of the logistics domain of the previous examples for execution on-line and handling of exogenous delivery requests at run-time. We also show some sample runs using a ConGolog interpreter in Prolog.

On-line execution of a ConGolog program means that once the first primitive action is determined according to the control structure of the program, which due to nondeterminism may involve randomly choosing one, this action is actually executed in the world. This means that our ConGolog interpreter should not backtrack after choosing such an action. Luckily, this behaviour is very easy to realize in Prolog using a cut. The off-line interpreter includes the rule:

```
offline(Prog,S0,Sf):-
           final(Prog,S0), S0=Sf ;
           trans(Prog,S0,Prog1,S1),
           offline(Prog1,S1,Sf).
```

To prevent the interpreter from backtracking on primitive actions, including exogenous ones, we simply add a cut after a one step transition is chosen:

```
online(Prog,S0,Sf):-
          final(Prog,S0), S0=Sf ;
          trans(Prog,S0,Prog1,S1), !,
          online(Prog1,S1,Sf).
```

This is a *brave* online interpreter. A *cautious* one may, for instance, check offline that the remainder of the program successfully terminates before committing to an action:

```
online(Prog,S0,Sf):-
          final(Prog,S0), S0=Sf ;
          trans(Prog,S0,Prog1,S1),
          offline(Prog1,S1,Soff), !,
          online(Prog1,S1,Sf).
```

These issues are further discussed in (De Giacomo, Reiter, & Soutchanski 1998; Reiter 2001).

Let us now turn to exogenous actions. Although an agent, or in our case the logistics program, does not have control over when exogenous actions occur, its background theory includes axioms informing it what exogenous actions can occur and what their effects are. In our logistics example, we only consider one exogenous action: $requestDelivery(obj, loc)$ meaning that a request to deliver $obj$ to $loc$ has been issued. Exogenous actions will be generated by having the interpreter ask the user to input them.

Following (De Giacomo, Lesperance, & Levesque 2000), we will model exogenous actions by defining a special procedure which will execute in parallel with the logistics main procedure:

$$\begin{array}{l}\textbf{proc}\ exoProg\\ \quad (\pi e)(exoActionOccurred(e) \rightarrow e)\\ \textbf{endProc}\end{array}$$

The condition $exoActionOccurred(e)$ always succeeds when evaluated and it comes back with a user supplied value for $e$ which can be an exogenous action, $nil$ which means no exogenous action occurred, or $endSim$ which is just as $nil$ but tells the interpreter to stop asking the user for exogenous actions. We could alternatively have had them generated randomly without complication.

Now, the main logistics procedure is a program that reacts to the occurrence of exogenous actions $requestDelivery(obj, loc)$ by triggering the execution of a $moveObj(obj, loc)$ task:

$$\begin{array}{l}\textbf{proc}\ deliveryDaemon\\ \quad (\pi pck, loc)\ deliveryReq(pck, loc) \rightarrow\\ \quad\quad startDelivery(pck, loc)\ ;\\ \quad\quad [(moveObj(pck, loc)\ ;\\ \quad\quad\quad endDelivery(pck, loc)) \parallel\\ \quad\quad\ deliveryDaemon]\\ \textbf{endProc}\end{array}$$

The main ConGolog program is the parallel execution of the logistics procedure and the exogenous actions procedure: $exoProg \parallel deliveryDaemon$.

Here is a sample run in Eclipse Prolog:

```
[eclipse 2]: runSim.
startSim
  Enter an exogenous action:
        requestDelivery(package1, loc5-1).
requestDelivery(package1, loc5-1)
startDelivery(package1, loc5-1)
  Enter an exogenous action:  nil.
driveTruck(truck3-1, loc3-1, loc3-3)
  Enter an exogenous action:  nil.
loadTruck(package1, truck3-1)
  Enter an exogenous action:  nil.
driveTruck(truck3-1, loc3-3, loc3-1)
unloadTruck(package1, truck3-1)
  Enter an exogenous action:  nil.
fly(plane1, loc5-1, loc3-1)
  Enter an exogenous action:
  requestDelivery(package2, loc3-2).
requestDelivery(package2, loc3-2)
loadAirplane(package1, plane1)
fly(plane1, loc3-1, loc5-1)
unloadAirplane(package1, plane1)
startDelivery(package2, loc3-2)
  Enter an exogenous action:  nil.
endDelivery(package1, loc5-1)
loadTruck(package2, truck3-1)
driveTruck(truck3-1, loc3-1, loc3-2)
```

```
unloadTruck(package2, truck3-1)
  Enter an exogenous action:
  requestDelivery(package3, loc1-3).
requestDelivery(package3, loc1-3)
  Enter an exogenous action:  nil.
startDelivery(package3, loc1-3)
endDelivery(package2, loc3-2)
driveTruck(truck2-1, loc2-1, loc2-3)
  Enter an exogenous action:  nil.
loadTruck(package3, truck2-1)
driveTruck(truck2-1, loc2-3, loc2-1)
unloadTruck(package3, truck2-1)
  Enter an exogenous action:  nil.
loadAirplane(package3, plane2)
  Enter an exogenous action:  nil.
fly(plane2, loc2-1, loc1-1)
  Enter an exogenous action:  nil.
  Enter an exogenous action:  endSim.
endSim
unloadAirplane(package3, plane2)
loadTruck(package3, truck1-1)
driveTruck(truck1-1, loc1-1, loc1-3)
unloadTruck(package3, truck1-1)
endDelivery(package3, loc1-3)

Plan length: 32 More?  n.
```

The non-indented lines are primitive tasks appearing in the order they occur. The user is prompted for an exogenous action every time the condition $exoActionOccurred(e)$ is evaluated. This happens every time the interpreter computes a transition for the $exoProg$ procedure.

## Conclusion

Our purpose was two-fold. On one hand we have argued that HTN-planning can be thought of as a special case of high-level programming in the sense of Golog/ConGolog. We have done this by showing an encoding of HTN-planning problems in these languages. In doing this, we only took advantage of a few of their constructs and of the techniques which have been developed for the many problems that have arisen in cognitive robotics research. These techniques are obviously relevant to planning given that both problems involve modeling dynamic worlds. The work by the Cognitive Robotics group at the U. of Toronto includes formalizations for robotic control that account for explicit time of action occurrence, sensing and knowledge, execution monitoring, stochastic actions, action choice based on decision theory, and others.[3] Our second goal was to actually show a generalization of HTN-planning, after taking this programming perspective, by taking a classic HTN-planning problem, a logistics domain problem, and encoding it in ConGolog for on-line execution and run-time exogenous actions.

We were not the first to point out a connection between HTN-planning and high-level languages Golog and ConGolog. Baral and Son (1999) extended ConGolog with an HTN construct. In the extended language, a program may include an HTN-planning problem as a statement. However, the new construct is limited: the tasks appearing in it cannot be ConGolog programs. One has to separately define methods for the compound tasks mentioned in an HTN-statement.

[3] Much of this work can be found at http://www.cs.toronto.edu/cogrobo

## Acknowledgments

We are thankful to Ray Reiter and Fahiem Bacchus for helpful discussions on the subject of this paper.

## References

Bacchus, F., and Kabanza, F. 1995. Using temporal logic to control search in a forward chaining planner. In *Proceedings of the Third European Workshop on Planning*.

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 16:123–191.

Baral, C., and Son, T. C. 1999. Extending ConGolog to allow partial ordering. In *Proc. of the Sixth International Workshop on Agent Theories, Architectures, and Languages (ATAL-99)*, volume 1757 of *LNCS*, 188–204.

De Giacomo, G.; Lesperance, Y.; and Levesque, H. J. 2000. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence* 121:109–169.

De Giacomo, G.; Reiter, R.; and Soutchanski, M. 1998. Execution monitoring of high-level robot programs. In *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*.

Erol, K.; Hendler, J. A.; and Nau, D. S. 1996. Complexity results for hierarchical task-network planning. *Annals of Mathematics and Artificial Intelligence* 18:69–93.

Levesque, H. J.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. B. 1997. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming* 31(1–3):59–83.

Levesque, H.; Pirri, F.; and Reiter, R. 1998. Foundations for the situation calculus. *Linköping Electronic Articles in Computer and Information Science* 3(18). http://www.ep.liu.se/ea/cis/1998/018/.

McCarthy, J. 1963. Situations, actions and causal laws. Technical report, Stanford University. Reprinted in Semantic Information Processing (M. Minsky ed.), MIT Press, Cambridge, Mass., 1968, pp. 410–417.

Nau, D. S.; Cao, Y.; Lotem, A.; and Munoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, 968–975.

Reiter, R. 1991. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Lifschitz, V., ed., *Artificial Intelligence and Mathematical Theory of Computation*. Academic Press. 359–380.

Reiter, R. 2001. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems*. Cambridge, MA: MIT Press.

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