Extending ConGolog to allow partial ordering

Chitta Baral[*] Tran Cao Son[†]

[*] Department of Computer Science and Engineering, Arizona State University
Tempe, AZ 85287, USA
chitta@asu.edu

[†] Department of Computer Science, University of Texas at El Paso
El Paso, Texas 79968, USA
tson@cs.utep.edu

Abstract. In this paper we extend the high level execution language *ConGolog* (developed at the University of Toronto) by adding to it a new construct which we call the *htn*-construct. The new construct improves *ConGolog* by allowing easy specification of non-determinism when a partial ordering between a set of actions needs to be maintained. Furthermore, it allows temporal constraints to be specified easily. We present an implementation of the *htn*-construct in PROLOG which can be directly added to PROLOG implementations of ConGolog interpreters.

1 Introduction

Golog [8] and *ConGolog* [3–5] are two main *high level execution languages* developed recently (in the 90's) by the Cognitive Robotics group at the University of Toronto. The approach to write programs in these languages for high level control of robots and other agents in a dynamic and incompletely specified world is proposed as an alternative to the traditional approach based on plan synthesis.

The approach of writing programs in *Golog/ConGolog* exploits the fact that often the people involved in describing the planning domain have some ideas about the plan they are looking for. In the plan synthesis approach these ideas can be used as domain specific knowledge to make it easier to find a plan. In the *Golog/ConGolog* approach these ideas are made part of the plan and non-deterministic constructs are used to encode those part of the plan that the domain encoder is not sure of. The non-determinism is then sorted out by the *Golog/ConGolog* interpreter by doing the necessary searching and reasoning.

But much before *Golog* and *ConGolog*, hierarchical task networks (or HTNs) [14] have been proposed as a planning language, whose basic philosophy is very similar to that of *Golog/ConGolog*, and since then they have been used extensively in planning. A HTN-planner and an offline *Golog/ConGolog* interpreter are similar in that they both take a (usually incomplete specified) plan as input and produces a complete plan as output. But there are subtle differences between them and each has some strong points. We first list some of the features of HTN that are absent in *Golog/ConGolog* and then the opposite.

Strong points of HTN

(i) It is straight forward to express partial ordering between actions in HTN. In comparison the non-deterministic constructs in *Golog/ConGolog* are limited and do not allow us to easily specify a partial order between a set of actions and let the interpreter pick a particular total order. Such a construct is very important and used in most HTN plans. A simple example of its use is the action of taking classes to obtain a degree. The pre-requisite relation between various courses forms a partial order.

(ii) Besides allowing pre and post conditions, HTNs also allow the specification of particular kind of temporal conditions, where fluents are required to hold between two (not necessarily consecutive) action steps.

(iii) Finally, since HTNs have been used extensively in real planning systems, it is perhaps important that the later execution languages are upward compatible with HTNs; i.e., they have HTN constructs in them.

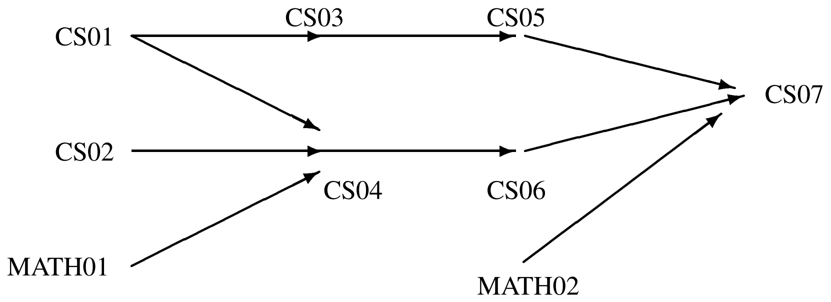
Some strong points of *Golog/ConGolog*

(a) They allow ALGOL-style constructs, such as, sequences, and loops. These constructs can be simulated in HTN, but in a round about way. It is more natural to use the ALGOL-style constructs.

(b) *Golog/ConGolog* use a sophisticated logic of actions (based on Situation Calculus), which allows a rich language to specify the effects of actions and constraints about the world, and can reason with incomplete information about the world. In contrast, in HTN the language to specify the effects of actions is extremely simple and limited. For example, it does not allow specification of domain constraints or causal constraints. In addition it requires that we have complete knowledge about the initial situation.

We illustrate the differences of HTN-representation and *Golog/ConGolog* representation in the next example.

Example 1. Suppose that we would like to create a study plan for a certain degree. The rules say that one needs to take all the classes *CS01, CS02, CS03, CS04, CS05, CS06, MATH01,* and *MATH02*. The rules also contain prerequisite information of classes. This information is depicted in the next graph where an arrow from class *X* to class *Y* means that *X* is a prerequisite of *Y*.



The above information can be represented using an action theory in which an action of taking a class, say $CS01$, will cause the class to be taken, i.e., $taken(CS01)$ becomes true. As such, creating a study plan is equivalent to finding a plan achieving the goal $taken(CS07)$. Let us assume that we have an action theory for the domain and we would like to use a HTN-planner or a *Golog/ConGolog* interpreter to construct the plan. To do it properly, we need to represent the prerequisite relationship between classes in both formalisms. The domain specific information can be encoded in HTN and *Golog/ConGolog* as follows.

- *HTN-representation*
  $CS01 < CS03 < CS05 < CS07$,
  $CS01, CS02 < CS04 < CS06 < CS07$,
  $MATH01 < CS04$,$MATH02 < CS07$

- *ConGolog-program*
  $((((CS01; CS03; CS05)\|$
  $(CS02\|MATH01)); taken(CS01)?; CS04; CS06)\|MATH02);$
  $CS07$

While it is rather straightforward to maps the prerequisite ordering into HTN-constraints, it is not easy to encode this information into a *ConGolog*-program. Moreover, it is difficult to recognize the ordering information from the *ConGolog*-program. On the other hand, using *ConGolog* we can easily say that some actions can be executed in parallel, say $CS02$ can be taken at the same time with $MATH01$, which is much less trivial in the HTN-representation. □

Our aim in this paper is to take advantage of the features of both HTN and Golog/ConGolog *and overcome their limitations.* To do this we propose to add an HTN construct to *ConGolog*. We refer to this language as *ConGolog*+HTN and present a logic programming interpreter for it. (Note that all the *Golog/ConGolog* interpreters that we know of are written in Prolog.)

2 Preliminaries

In this section we review the basic definitions of *ConGolog* and HTN. We first begin with the basics of the situation calculus language [10].

2.1 Situation Calculus

The basic components of the situation calculus language in the notation of [13][1] include a special constant S_0 denoting the initial situation, a binary function symbol do where $do(a, s)$ denotes the successor situation to s resulting from executing the action

[1] See [13] for detailed description of the situation calculus language.

a, $fluent$ relations of the form $f(s)$, denoting that the fluent f is true in the situation s, [2] and a special predicate $Poss(a, s)$ denoting that action a is executable in situation s.

A dynamic domain can then be represented by a theory comprising of axioms describing the initial situation S_0; action precondition axioms (one for each primitive action a, characterizing $Poss(a, s)$); successor state axioms (one for each fluent F, stating under what condition $F(\boldsymbol{x}, do(a, s))$ holds, as a function of what holds in s); Unique names axioms for the primitive actions; and some foundational, domain independent axioms.

2.2 ConGolog

The constructs of *ConGolog* [3–5] are:

α	primitive action
$\phi?$	wait for a condition
$(\sigma_1 ; \sigma_2)$	sequence
$(\sigma_1 \vert \sigma_2)$	nondeterministic choice between actions
$\pi x.\sigma$	nondeterministic choice of arguments
σ^*	nondeterministic iteration
if ϕ **then** σ_1 **else** σ_2	synchronized conditional
while ϕ **do** σ	synchronized loop
proc $\beta(\boldsymbol{x})\sigma$	procedure definition
$(\delta_1 \vert\vert \delta_2)$	concurrent execution
$(\delta_1 \rangle\rangle \delta_2)$	concurrency with different priorities
$\delta^{\vert\vert}$	concurrent iteration
$< \phi \rightarrow \delta >$	interrupt

The original semantics of *Golog* was given by a formula $Do(\delta, s, s')$ where δ is a program, and s and s' are situations. Intuitively, $Do(\delta, s, s')$ holds whenever the situation s' is a terminating situation of an execution of δ starting from s. In [3], it is suggested that for more complicated constructs such as concurrency, interrupts, etc., it is better to define the semantics in terms of two predicates $Final$ and $Trans$ where $Final(\sigma, s)$ means that the program σ may legally terminate in the situation s and $Trans(\sigma, s, \sigma', s')$ means that program σ in situation s may legally execute one step, ending in the situation s' with program σ' remaining. Some of the axioms for $Trans$ and $Final$, taken from [3–5], are given below.

$Final(nil, s) \equiv TRUE$
$Final(\alpha, s) \equiv FALSE$
$Final(\phi?, s) \equiv FALSE$
$Final(\sigma_1; \sigma_2, s) \equiv Final(\sigma_1, s) \wedge Final(\sigma_2, s)$
$Final(\sigma_1 | \sigma_2, s) \equiv Final(\sigma_1, s) \vee Final(\sigma_2, s)$
$Final(\textbf{if } \phi \textbf{ then } \sigma_1 \textbf{ else } \sigma_2, s) \equiv \phi(s) \wedge Final(\sigma_1, s) \vee \neg\phi(s) \wedge Final(\sigma_2, s)$
$Final(\textbf{while } \phi \textbf{ do } \sigma, s) \equiv Final(\sigma, s) \vee \neg\phi(s)$
$Final(\sigma_1 || \sigma_2, s) \equiv Final(\sigma_1, s) \wedge Final(\sigma_2, s)$ $Final(\pi v.\sigma, s) \equiv \exists x Final(\sigma^v_x, s)$

[2] In our logic programming implementation, we use a slightly different notation; instead of saying $f(s)$, where f is a fluent, we write $holds(f, s)$.

$Final(\sigma^*, s) \equiv TRUE$
$Final(\sigma_1 \langle\langle \sigma_2, s) \equiv Final(\sigma_1, s) \wedge Final(\sigma_2, s)$
$Final(\sigma^{||}, s) \equiv True$

$Trans(nil, s, \delta, s') \equiv FALSE$
$Trans(\alpha, s, \delta, s') \equiv Poss(\alpha, s) \wedge \delta = nil \wedge s' = do(\alpha, s)$
$Trans(\phi?, s, \delta, s') \equiv \phi(s) \wedge \delta = nil \wedge s = s'$
$Trans(\sigma_1; \sigma_2, s, \delta, s') \equiv$
$\quad Final(\sigma_1, s) \wedge Trans(\sigma_2, s, \delta, s') \vee \exists \delta'.\delta = (\delta'; \sigma_2) \wedge Trans(\sigma_1, s, \delta', s')$
$Trans(\sigma_1 | \sigma_2, s, \delta, s') \equiv Trans(\sigma_1, s, \delta, s') \vee Trans(\sigma_2, s, \delta, s')$
$Trans(\sigma^*, s, \delta, s') \equiv \exists \delta'.\delta = (\delta'; \sigma^*) \wedge Trans(\sigma, s, \delta', s')$
$Trans(\textbf{if } \phi \textbf{ then } \sigma_1 \textbf{ else } \sigma_2, s, \delta, s') \equiv$
$\quad \phi(s) \wedge Trans(\sigma_1, s, \delta, s') \vee \neg\phi(s) \wedge Trans(\sigma_2, s, \delta, s')$
$Trans(\textbf{while } \phi \textbf{ do } \sigma, s, \delta, s') \equiv$
$\quad \phi(s) \wedge \exists \delta'.(\delta = \delta'; \textbf{ while } \phi \textbf{ do } \sigma) \wedge Trans(\delta, s, \delta', s')$
$Tran(\sigma_1 || \sigma_2, s, \delta, s') \equiv$
$\quad \exists \delta'.\delta = (\delta' || \sigma_2) \wedge Trans(\sigma_1, s, \delta', s') \vee \exists \delta'.\delta = (\sigma_1 || \delta') \wedge Trans(\sigma_2, s, \delta', s')$
$Trans(\pi v.\sigma, s, \delta, s') \equiv \exists x Trans(\sigma^v_x, s, \delta, s')$
$Tran(\sigma_1 \langle\langle \sigma_2, s, \delta, s') \equiv$
$\quad \exists \delta'.\delta = (\delta' \langle\langle \sigma_2) \wedge Trans(\sigma_1, s, \delta', s') \vee$
$\quad \exists \delta'.\delta = (\sigma_1 \langle\langle \delta') \wedge Trans(\sigma_2, s, \delta', s') \wedge \neg\exists \gamma, s''.Trans(\delta_1, s, \gamma, s'')$
$Trans(\sigma^{||}, s, \delta, s') \equiv \exists \gamma.\delta = (\gamma || \sigma_{||}) \wedge Trans(\delta, s, \gamma, s')$

With the help of $Trans$ and $Final$, $Do(\sigma, s, s')$ is defined by

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

where $Trans^*$ is the transitive closure of $Trans$, defined as the second-order situation calculus formula:

$$\begin{aligned} Trans^*(\sigma, s, \sigma', s') \stackrel{def}{=} & \forall T[\forall s.T(\sigma, s, \sigma, s) \wedge \\ & \forall s, \delta', s', \delta''.T(\sigma, s, \delta', s') \wedge \\ & \quad Trans(\delta', s', \delta'', s'') \supset T(\sigma, s, \delta'', s'')] \end{aligned}$$

2.3 Hierarchical Task Networks

We now give an overview about HTN. For our purpose, we do not strictly follow the HTN language in [1]. Rather, we formulate it using the terminologies of the situation calculus language. Throughout this section, by $\mathcal{D}$ we denote a situation calculus domain with its actions, constants, variables, etc. Actions in $\mathcal{D}$ are called *primitive actions*. We will assume that in addition to the domain $\mathcal{D}$, we have an infinite set of names N and an infinite set of compound task symbols T.

For a primitive action a of $\mathcal{D}$, $do\_p(a(\boldsymbol{x}))$[3] denotes a *primitive task*. A *goal task* has the form $achieve(l)$, where l is a fluent literal. And a *compound task* has the form

[3] To avoid confusion with the do function in situation calculus, instead of the original notation of $do(a(\boldsymbol{x}))$ in the HTN literature, we use $do\_p(a(\boldsymbol{x}))$ to denote a primitive task.

$perform[t(\boldsymbol{x})]$ where t is a compound task symbol in T. Here, $\boldsymbol{x}$ denotes the parameters of a primitive action or a compound task. We will often omit this parameter in the rest of the paper.

A *labeled task* is a pair (n, α) where n is a name and α is a primitive task, goal task, or a compound task. A *set of tasks* d is denoted by a sequence (PROLOG-list) $[(n_1, \alpha_1), \ldots, (n_k, \alpha_k)]$ where the (n_i, α_i)'s are labeled tasks.

A *task constraint* is a Boolean formula ϕ constructed from:

(i) variable binding constraints such as $v = v'$ and $v = c$,
(ii) ordering constraints such as $n \prec n'$, and
(iii) temporal constraints[4] such as (n, l), (l, n), and (n, l, n'),

where n, n' are task names, v, v' are variables, l is a fluent literal, and c is a constant. $n \prec n'$ means that the task labeled with n precedes the one labeled with n'; and (n, l), (l, n), and (n, l, n'), mean that l needs to be true immediately after n, immediately before n, and between n and n' respectively.

A *task network* is a pair (d, ϕ) where d is a set of tasks and ϕ is a task constraint.

A *method* is a triple $(name, d, \phi)$ where $name$ is a compound task symbol, and (d, ϕ) is a task network. Intuitively, a method describes what needs to be done in order to get the compound task $name$ done. A method could be viewed as a procedure in *ConGolog*.

An HTN planning domain is given by a pair $(\mathcal{D}, Methods)$ where $\mathcal{D}$ is a domain description and $Methods$ is a set of methods.

A planning problem is a tuple $\mathbf{P} = \langle d, \phi, s, \mathcal{D}, Methods \rangle$ where (d, ϕ) is a task network, s is a situation and $(\mathcal{D}, Methods)$ is a planning domain.

A *solution* to a planning problem $\mathbf{P} = \langle d, \phi, s, \mathcal{D}, Methods \rangle$ is defined inductively as follows.

Let (d, ϕ) be a primitive task network that contains only primitive tasks and let S_0 be the initial situation. A sequence of actions (or a plan) σ is a *completion* of (d, ϕ) at S_0, denoted by $\sigma \in comp(d, \phi, S_0, \mathcal{D})$, if σ is a total ordering of the primitive tasks in a ground instance of d that is executable in S_0 and satisfies the constraint formula ϕ.

Let d be a non-primitive task network that contains a (non-primitive) task (n, α). Let $m = (\alpha', d', \phi')$ be a method, and η be the most general unifier of α and α'. Then we define $reduced(d, \phi, n, m)$ to be the task network obtained from $d\eta$ be replacing $(n, \alpha)\eta$ with the task nodes of $(d', \phi')\eta$, and incorporating $d'\eta$'s constraints formula into the constraint formula ϕ. We denote the set of reductions of (d, ϕ) by $red(d, \phi, S_0, \mathcal{D})$. Let

$$sol_1(d, \phi, S_0, \mathcal{D}) = \begin{cases} comp(d, \phi, S_0, \mathcal{D}) \text{ if } d \text{ is a primitive task network} \\ \emptyset \text{ otherwise} \end{cases}$$

[4] In [1], it is called *truth constraints*.

and $sol_{n+1}(d, \phi, S_0, \mathcal{D}) = sol_n(d, \phi, S_0, \mathcal{D}) \cup \bigcup_{d' \in red(d,\phi,S_0,\mathcal{D})} sol_n(d', \phi, S_0, \mathcal{D})$.

It has been proved in [1] that $sol(d, \phi, S_0, \mathcal{D}) = \bigcup_{n<\omega} sol_n(d, \phi, S_0, \mathcal{D})$ is the set of solutions to the problem **P**.

3 ConGolog+HTN

3.1 The *htn*-construct

To add to *ConGolog* the features of HTN, we add to *ConGolog* an htn construct. An htn-program is specified by a formula of the form $htn(d, \phi)$ where d is a task sequence and ϕ is a task constraint. An htn-program could be considered as a complex action of *ConGolog*. It can be placed anywhere a *ConGolog* -procedure call can be.

To complete the integration of the htn-construct into *ConGolog* we need to give the axioms for $Trans$ and $Final$ for htn-programs. Before doing that let us define some predicates which we will be using in defining $Trans$. Let d be a task sequence and ϕ be a task constraint[5]. We define $primitive/1$, $comp/3$, and $reduces/2$ to be the predicates with the following intended meaning.

(i) $primitive(d)$ holds if d is a sequence of primitive tasks;
(ii) $comp(htn(d, \phi), \alpha, s)$ is defined when $primitive(d)$ holds. It is intended to say that α is a sequence of all actions in d such that the total ordering induced by α makes ϕ satisfiable if α is executed in s.
(iii) $reduces(htn(d, \phi), htn(d', \phi'))$ is defined when d contains some compound task. Intuitively, it means that $htn(d', \phi')$ is obtained from $htn(d, \phi)$ by selecting a compound task (n, α) from d and updating d and ϕ using a method for α.

The axioms for $Trans$ and $Final$ for htn-programs are given below.

$$
\begin{array}{ll}
Final(htn([], \phi), s) & \equiv (\phi \equiv TRUE) \\
Final(htn(d, \phi), s) & \equiv FALSE \\
Trans(htn(d, \phi), s, \sigma, s') & \equiv [primitive(d) \wedge \\
& \qquad \exists \alpha.[comp(htn(d, \phi), \alpha, s) \wedge Trans(\alpha, s, \sigma, s')]] \vee \\
& \quad [\neg primitive(d) \wedge \\
& \qquad \exists d', \phi'.[reduces(htn(d, \phi), htn(d', \phi')) \wedge \\
& \qquad\qquad Trans(htn(d', \phi'), s, \sigma, s')]]
\end{array}
$$

We omit here the classical logic definition of $primitive$, $comp$, and $reduces$. Rather we present their logic programming implementation next.

[5] We simplify the notation by not mentioning other components of the planning domain in which d and ϕ are defined. We assume that this will be clear from the context. Whenever it is needed we will specify them explicitly.

3.2 Logic Programming Implementation

We now present an implementation of the htn-construct in PROLOG, which can be added to any *ConGolog* implementation. We will assume that:

- methods are given by the predicate $method(task\_name, d, \phi)$;
- actions in $\mathcal{D}$ are listed by the predicate $action(action\_name)$; and
- fluent literals are specified by the predicate $literal(fluent\_name)$.

We now present two interpreters for htn-programs: an *offline* interpreter and a cautious *online* interpreter. We should mention that these interpreters will not run as stand-alone applications. Rather, they have to be used in conjunction with the offline or online interpreter of *ConGolog*. Both of our interpreters use some well-known library predicates of PROLOG such as $member$, $permutation$, is_list etc. and share the following auxiliary predicates:

(i) $name\_list/2$, where $name\_list(list, d)$ holds if $list$ is the list of task names in the list of tasks d;
(ii) $is\_true/1$ where $is\_true(\phi)$ holds if ϕ, a Boolean formula, is evaluated to be true.

Offline Interpreter Let d be a sequence of primitive tasks, s be a situation of our domain description – which we represent by the sequence of actions executed from the initial situation S_0 to reach s – and ϕ be a task constraint whose constraints can contain a list of names instead of a single name as defined earlier. For example, it might contain some ordering constraints of the form $[n_1, \ldots, n_k] \prec n$ or $n \prec [n_1, \ldots, n_k]$ where n and n_j's are names. Let $nlist$ be a list of names. The following predicates are defined only for primitive task sequences.

(i) $first\_of/3$ where $first\_of(nlist, d, n)$ holds if n is the name of the first task in the list of tasks d among the names in the list $nlist$; $last\_of/3$ is defined similarly;
(ii) $first\_name/2$ where $first\_name(d, name)$ holds if $name$ is the name of the first task (w.r.t. the list ordering) of d;
(iii) $upd\_name/4$ where $upd\_name(nl', n, nl, nl'')$ holds if nl'' (a list of names or a single name) is obtained by replacing n (a name) in nl' (a list of names, or a single name) by nl and flattening the resulting list;
(iv) $instance/2$ where $instance(d, \alpha)$ holds if d is a list of primitive tasks and α is a list of all the actions in d;
(v) $ordered/2$ where $ordered(n < n', d)$ holds if the ordering constraint $n \prec n'$ (n and n' are task names in d) is satisfied by the task sequence d;
(vi) $is\_ordered/2$ where $is\_ordered(n < n', d)$ holds if the ordering constraint $n \prec n'$ (n and n' can be list of names in d) is satisfied by the task sequence d;
(vii) $after/4$ where $after(n, l, d, s)$ holds if l holds after executing the task (n, α) of d in s;
(viii) $before/4$ where $before(n, l, d, s)$ holds if l holds in s;
(ix) $between/5$ where $between(n, l, n', d, s)$ holds if l holds between the executions of the tasks (n, α) and (n', α') of d when d is executed sequentially (w.r.t. the list order) in s;

(x) $sat/3$ where $sat(\phi, d, s)$ holds if the task sequence d (and the total ordering induced by it) would satisfy the constraint ϕ if it were to be executed in s;

(xi) $update/4$ where $update(\phi, n, list, \phi')$ holds if the constraint ϕ' is obtained from the constraint ϕ by replacing every occurrence of a task name n with a list of task names $list$.

We omit here the rules for $first\_of(nlist, d, n)$, $last\_of(nlist, d, n)$, $first\_name$, and $instance$ which are rather straightforward. Other predicates are defined as follows.

$$
\begin{array}{rl}
ordered(n_1 < n_2, [(n_1, \_)|d]) \leftarrow & member((n_2, \_), d). \\
ordered(n_1 < n_2, [(n_3, \_)|d]) \leftarrow & n_1 \neq n_3, \\
& ordered(n_1 < n_2, d). \\
is\_ordered(n_1 < n_2, d) \leftarrow & last\_of(n_1, d, n_3), \\
& first\_of(n_2, d, n_4), \\
& ordered(n_3 < n_4, d).
\end{array} \tag{1}
$$

$$
\begin{array}{rl}
after(n, l, [(n, do\_p(a))|d], s) \leftarrow & executable(a, s), \\
& holds(l, [a|s]). \\
after(n, l, [(n_1, do\_p(a))|d], s) \leftarrow & n \neq n_1, \\
& executable(a, s), \\
& after(n, l, d, [a|s]). \\
after(n, l, [(n_1, achieve(l_1))|d], s) \leftarrow & holds(l_1, s), \\
& after(n, l, d, s).
\end{array} \tag{2}
$$

$$
\begin{array}{rl}
before(n, l, [(n, do\_p(a))|d], s) \leftarrow & executable(a, s), \\
& holds(l, s). \\
before(n, l, [(n_1, do\_p(a))|d], s) \leftarrow & n \neq n_1, \\
& executable(a, s), \\
& before(n, l, d, [a|s]). \\
before(n, l, [(n_1, achieve(l_1))|d], s) \leftarrow & holds(l_1, s), \\
& before(n, l, d, s).
\end{array} \tag{3}
$$

$$
\begin{array}{rl}
between(n, l, n, d, s) \leftarrow & \\
between(n, l, n_2, [(n, do\_p(a))|d], s) \leftarrow & executable(a, s), \\
& holds(l, [a|s]), \\
& first\_name(d, n_3), \\
& between(n_3, l, n_2, d, [a|s]). \\
between(n_1, l, n_2, [(n, do\_p(a))|d], s) \leftarrow & executable(a, s), \\
& n \neq n_1, n \neq n_2, \\
& between(n_1, l, n_2, d, [a|s]). \\
between(n_1, l, n_2, [(n, achieve(l_1))|d], s) \leftarrow & holds(l_1, s), \\
& between(n_1, l, n_2, d, s).
\end{array} \tag{4}
$$

where $holds(f, s)$ is the predicate asserting that the fluent literal f holds in s. $executable(a, s)$ indicates that a is executable in s. [6]

[6] In this paper, we are more interested in executing a htn-program. And hence, goal task of the form $(n, achieve(l))$ is treated simply as a node where l must be satisfied. This is similar to

Using the above predicates we can define the predicate $sat(\phi, d, s)$. The first group of rules for $sat(\phi, d, s)$ deals with individual constraints.

$$\begin{aligned}
sat(n_1 < n_2, d, s) &\leftarrow is_ordered(n_1 < n_2, d).\\
sat((n, l), d, s) &\leftarrow last_of(n, d, n_1),\\
&\quad\; after(n_1, l, d, s).\\
sat((l, n), d, s) &\leftarrow first_of(n, d, n_1),\\
&\quad\; before(n_1, l, d, s).\\
sat((n_1, l, n_2), d, s) &\leftarrow last_of(n_1, d, n_3),\\
&\quad\; first_of(n_2, d, n_4),\\
&\quad\; between(n_3, l, n_4), d, s).
\end{aligned} \tag{5}$$

The next group of rules of $sat(\phi, d, s)$ deals with formula of constraints.

$$\begin{aligned}
sat(\phi_1 \wedge \phi_2, d, s) &\leftarrow sat(\phi_1, d, s), sat(\phi_2, d, s).\\
sat(\phi_1 \vee \phi_2, d, s) &\leftarrow sat(\phi_1, d, s).\\
sat(\phi_1 \vee \phi_2, d, s) &\leftarrow sat(\phi_2, d, s).\\
sat(\neg\phi, d, s) &\leftarrow not\ sat(\phi, d, s).
\end{aligned} \tag{6}$$

We are ready to define the $comp$ predicate.

$$\begin{aligned}
comp(htn(d, \phi), \alpha, s) \leftarrow\; & permutation(d, d_1),\\
& sat(\phi, d_1, s),\\
& instance(d_1, \alpha).
\end{aligned} \tag{7}$$

To complete the definition of the offline interpreter, we now define the predicate $reduces$. Let $(n, perform(\alpha))$ be a compound task of (d, ϕ) and (d', ϕ') be a method of α. Updating the task (n, α) will create a new task network (d'', ϕ'')[7] as follows.

- $d'' = d \cup d' \setminus \{(n, \alpha)\}$;
- if $n \prec n'$ (resp. $n' \prec n$) belongs to ϕ, then $name\_list(d') \prec n'$ (resp. $n' \prec name\_list(d')$) belongs to ϕ'';
- if (n, l), (l, n), (n, l, n'), or (n', l, n) belongs to ϕ, then $(name\_list(d'), l)$, $(l, name\_list(d'))$, $(name\_list(d'), l, n')$, or $(n', l, name\_list(d'))$ belongs to ϕ'' respectively.

The details of this process is encoded in two predicated *update* and *upd_name* as follows.

what is done in [1] where it is replaced with a task of the form $(n, do\_p(nothing))$ and the constraint (l, n) is added to the task constraint.

[7] Strictly speaking, (d'', ϕ'') may not be a task network in the normal sense as its constraint may contain some elements whose syntax does not match with what has defined earlier.

$$
\begin{aligned}
upd\_name(n_1, n, nlist, n_1) &\leftarrow not\ is\_list(n_1),\\
&\quad\; n \neq n_1.\\
upd\_name(n_1, n, nlist, nlist) &\leftarrow not\ is\_list(n_1),\\
&\quad\; n = n_1.\\
upd\_name(n_1, n, nlist, n_1) &\leftarrow is\_list(n_1),\\
&\quad\; not\ member(n, n_1).\\
upd\_name(n_1, n, nlist, n_2) &\leftarrow is\_list(n_1),\\
&\quad\; member(n, n_1),\\
&\quad\; append(n_1, nlist, n_2).\\
update(n_1 < n_2, n, nlist, n_3 < n_4) &\leftarrow upd\_name(n_1, n, nlist, n_3),\\
&\quad\; upd\_name(n_2, n, nlist, n_4).\\
update((n_1, l), n, nlist, (n_2, l)) &\leftarrow literal(l),\\
&\quad\; upd\_name(n_1, n, nlist, n_2).\\
update((l, n_1), n, nlist, (l, n_2)) &\leftarrow literal(l),\\
&\quad\; upd\_name(n_1, n, nlist, n_2).\\
update((n_1, l, n_2), n, nlist, (n_3, l, n_4)) &\leftarrow literal(l),\\
&\quad\; upd\_name(n_1, n, nlist, n_3),\\
&\quad\; upd\_name(n_2, n, nlist, n_4).\\
update(\phi_1 \wedge \phi_2, n, nlist, \phi_3 \wedge \phi_4) &\leftarrow update(\phi_1, n, nlist, \phi_3),\\
&\quad\; update(\phi_2, n, nlist, \phi_4).\\
update(\phi_1 \vee \phi_2, n, nlist, \phi_3 \vee \phi_4) &\leftarrow update(\phi_1, n, nlist, \phi_3),\\
&\quad\; update(\phi_2, n, nlist, \phi_4).\\
update(\neg\phi, n, nlist, \neg\phi_1) &\leftarrow update(\phi, n, nlist, \phi_1).
\end{aligned}
\tag{8}
$$

Rules for $reduces$ are given next.

$$
\begin{aligned}
reduces(htn(d, \phi), htn(d_1, \phi_1 \wedge \phi_2)) \leftarrow\ & member((n, perform(\alpha)), d),\\
& method(\alpha, d_2, \phi_2),\\
& delete(d, (n, perform(\alpha)), d_3),\\
& append(d_3, d_2, d_1),\\
& name\_list(nl, d_2),\\
& update(\phi, n, nl, \phi_1).
\end{aligned}
\tag{9}
$$

We are now ready to define $Final$ and $Trans$ in logic programming.

$$
\begin{aligned}
final(htn([], \phi), s) &\leftarrow is\_true(\phi).\\
trans(htn(d, \phi), s, \sigma, s_1) &\leftarrow primitive(d),\\
&\quad\; comp(htn(d, \phi), \alpha, s),\\
&\quad\; trans(\alpha, s, \sigma, s_1).\\
trans(htn(d, \phi), s, \sigma, s_1) &\leftarrow not\ primitive(d),\\
&\quad\; reduces(htn(d, \phi), htn(d_1, \phi_1)),\\
&\quad\; trans(htn(d_1, \phi_1), s, \sigma, s_1).
\end{aligned}
\tag{10}
$$

The main predicate for the offline interpreter *offline*$(htn(d, \phi), s, s')$, which says that a successful execution of $htn(d, \phi)$ in s might end in s'; is defined next.

$$
\begin{aligned}
\textit{offline}(htn(d,\phi),s,s) &\leftarrow final(htn(d,\phi),s). \\
\textit{offline}(htn(d,\phi),s,s_1) &\leftarrow \textit{not } final(htn(d,\phi),s), \\
& trans(htn(d,\phi),s,\sigma,s_2), \\
& \textit{offline}(\sigma,s_2,s_1).
\end{aligned} \tag{11}
$$

Recall that the rules for $trans$ and $final$ are only for the *htn*-construct. To run a HTN+*ConGolog* program, one need to add the rules (1)-(11) to an offline interpreter of *Golog/ConGolog*.

A cautious online interpreter An offline interpreter given above first reduces $htn(d,\phi)$ to a valid sequence of actions to be executed in the given situation s, and then executes them. In contrast a straight forward online interpreter will commit to transitions as they are done, and runs the risk of reaching dead ends. Such an on-line interpreter can be defined by adding the cut operator just after the $trans$ in the second rule, and replacing *offline* by $online$ throughout the two rules. We define a cautious online interpreter that quickly and optimistically checks (through the predicate *qc*) its choice of action that is to be executed next or that is to be reduced next, and commits to this choice if qc holds, and then updates (through a new predicate *upd*) the constraint to reflect commitment to the choice that is made. We now give the intuitive meaning of qc and upd.

(i) $qc(n,\alpha,\phi,s)$ holds if executing the task (n,α) in s; does not violate the constraint ϕ in an obvious way;

(ii) $upd(\phi,n,\alpha,s,\phi')$ holds if after executing the task (n,α) in s the constraint ϕ' remains to be satisfied;

(iii) $eq(m,n)$ holds when $m=n$ or $m=\{n\}$, and $in(n,m)$ holds when $m=n$ or n is a member of n.

In the following rules, by x' we denote either $x \setminus \{n\}$ or $\{x\} \setminus \{n\}$, depending on whether x is a name or a list of names. The rules for $qc/4$ are given below.

$$
\begin{aligned}
qc(n, \alpha, true, s) &\leftarrow \\
qc(n, \alpha, m < k, s) &\leftarrow eq(m, n). \\
qc(n, \alpha, m < k, s) &\leftarrow not\ eq(m, n), \\
&\quad\ not\ in(n, k). \\
qc(n, do\_p(a), (m, l), s) &\leftarrow eq(m, n), \\
&\quad\ holds(l, [a|s]). \\
qc(n, achieve(x), (m, l), s) &\leftarrow eq(m, n), \\
&\quad\ holds(l, s), \\
&\quad\ holds(x, s). \\
qc(n, perform(\alpha), (m, l), s) &\leftarrow \\
qc(n, \alpha, (m, l), s) &\leftarrow not\ eq(m, n). \\
qc(n, \alpha, (l, m), s) &\leftarrow in(n, m), \\
&\quad\ holds(l, s). \\
qc(n, \alpha, (l, m), s) &\leftarrow not\ in(n, m). \\
qc(n, \alpha, (m, l, k), s) &\leftarrow not\ eq(m, n), \\
&\quad\ not\ eq(k, n). \\
qc(n, do\_p(a), (m, l, k), s) &\leftarrow eq(m, n), \\
&\quad\ holds(l, [a|s]). \\
qc(n, achieve(x), (m, l, k), s) &\leftarrow eq(m, n), \\
&\quad\ holds(l, s), \\
&\quad\ holds(x, s). \\
qc(n, perform(\alpha), (m, l, k), s) &\leftarrow \\
qc(n, do\_p(a), (*, l, m), s) &\leftarrow holds(l, s), holds(l, [a|s]). \\
qc(n, achieve(x), (*, l, m), s) &\leftarrow holds(l, s), \\
&\quad\ holds(x, s). \\
qc(n, perform(\alpha), (*, l, k), s) &\leftarrow \\
qc(n, \alpha, \phi_1 \wedge \phi_2, s) &\leftarrow qc(n, \alpha, \phi_1, s), \\
&\quad\ qc(n, \alpha, \phi_2, s). \\
qc(n, \alpha, \phi_1 \vee \phi_2, s) &\leftarrow qc(n, \alpha, \phi_1, s). \\
qc(n, \alpha, \phi_1 \vee \phi_2, s) &\leftarrow qc(n, \alpha, \phi_2, s). \\
qc(n, \alpha, \neg\phi, s) &\leftarrow not\ qc(n, \alpha, \phi, s).
\end{aligned}
\tag{12}
$$

Rules for the update function *upd* are given below.

$$
\begin{aligned}
upd(true, n, \alpha, s, true) &\leftarrow \\
upd(false, n, \alpha, s, false) &\leftarrow \\
upd(\phi_1 \wedge \phi_2, n, \alpha, s, \phi_3 \wedge \phi_4) &\leftarrow upd(\phi_1, n, \alpha, s, \phi_3), \\
&\quad\ upd(\phi_2, n, \alpha, s, \phi_4). \\
upd(\phi_1 \vee \phi_2, n, \alpha, s, \phi_3 \vee \phi_4) &\leftarrow upd(\phi_1, n, \alpha, s, \phi_3), \\
&\quad\ upd(\phi_2, n, \alpha, s, \phi_4). \\
upd(\neg\phi, n, \alpha, s, \neg\phi_1) &\leftarrow upd(\phi, n, \alpha, s, \phi_1).
\end{aligned}
\tag{13}
$$

$$
\begin{aligned}
upd(m < k, n, \alpha, s, true) &\leftarrow eq(m,n).\\
upd(m < k, n, \alpha, s, m' < k) &\leftarrow not\ eq(m,n),\\
&\ not\ in(n,k).\\
upd(m < k, n, \alpha, s, false) &\leftarrow not\ eq(m,n),\\
&\ in(n,k)).\\
upd((m,l), n, do\_p(a), s, true) &\leftarrow eq(m,n),\\
&\ holds(l,[a|s]).\\
upd((m,l), n, achieve(x), s, true) &\leftarrow eq(m,n),\\
&\ holds(l,s),\\
&\ holds(x,s).\\
upd((m,l), n, do\_p(a), s, false) &\leftarrow eq(m,n),\\
&\ not\ holds(l,[a|s]).\\
upd((m,l), n, achieve(x), s, false) &\leftarrow eq(m,n),\\
&\ not\ holds(l,s).\\
upd((m,l), n, achieve(x), s, false) &\leftarrow eq(m,n),\\
&\ not\ holds(x,s).\\
upd((m,l), n, \alpha, s, (m',l)) &\leftarrow not\ eq(m,n).\\
upd((l,m), n, \alpha, s, true) &\leftarrow in(n,m),\\
&\ holds(l,s).\\
upd((l,m), n, \alpha, s, false) &\leftarrow in(n,m),\\
&\ not\ holds(l,s).\\
upd((l,m), n, \alpha, s, (l,m)) &\leftarrow not\ in(n,m).\\
upd((m,l,k), n, \alpha, s, (m',l,k')) &\leftarrow not\ eq(m,n),\\
&\ not\ eq(k,n).\\
upd((m,l,k), n, \alpha, s, false) &\leftarrow not\ eq(m,n),\\
&\ eq(k,n).\\
upd((m,l,k), n, do\_p(a), s, (*,l,k)) &\leftarrow eq(m,n),\\
&\ holds(l,[a|s]).\\
upd((m,l,k), n, achieve(x), s, (*,l,k)) &\leftarrow eq(m,n),\\
&\ holds(x,s),\\
&\ holds(l.s).\\
upd((m,l,k), n, do\_p(a), s, false) &\leftarrow eq(m,n),\\
&\ not\ holds(l,[a|s]).\\
upd((m,l,k), n, achieve(x), s, false) &\leftarrow eq(m,n),\\
&\ not\ holds(x,s).\\
upd((m,l,k), n, achieve(x), s, false) &\leftarrow eq(m,n),\\
&\ not\ holds(l,s).\\
upd((*,l,k), n, \alpha, s, true) &\leftarrow eq(k,n), holds(l,s).\\
upd((*,l,k), n, \alpha, s, (*,l,k')) &\leftarrow in(n,k),\\
&\ holds(l,s).\\
upd((*,l,k), n, \alpha, s, false) &\leftarrow in(n,k),\\
&\ not\ holds(l,s).
\end{aligned}
$$

We are now ready to define the predicate $on\_trans(htn(d, \phi), s, htn(d', \phi'), s')$ for the online interpreter. (Since we make a commitment after we pick an action, we need an on-line version of $trans$ to implement our cautious online interpreter. For a straightforward online interpreter we could have used the regular $trans$.)

$$
\begin{aligned}
on\_trans(htn(d, \phi), s, htn(d_1, \phi_1), [a|s]) \leftarrow\ & member((n, do\_p(a)), d), \\
& qc(n, do\_p(a), \phi, s), \\
& executable(a, s), \\
& delete(d, (n, do\_p(a)), d_1), \\
& upd(\phi, n, do\_p(a), s, \phi_1). \\
on\_trans(htn(d, \phi), s, htn(d_1, \phi_1), s) \leftarrow\ & member((n, achieve(l)), d), \\
& qc(n, achieve(l), \phi, s), \\
& delete(d, (n, achieve(l)), d_1), \\
& holds(l, s), \\
& upd(\phi, n, achieve(l), s, \phi_1). \\
on\_trans(htn(d, \phi), s, htn(d_1, \phi_1 \wedge \phi_2), s) \leftarrow\ & member((n, perform(\alpha)), d), \\
& qc(n, perform(\alpha), \phi, s), \\
& method(\alpha, d_2, \phi_2), \\
& delete(d, (n, perform(\alpha)), d_3), \\
& append(d_3, d_2, d_1), \\
& name\_list(nl, d_2), \\
& update(\phi, n, nl, \phi_1).
\end{aligned}
\tag{14}
$$

The main predicate for the online interpreter is defined next.

$$
\begin{aligned}
online(htn([], \phi), s, s) &\leftarrow is\_true(\phi). \\
online(htn(d, \phi), s, s_2) &\leftarrow on\_trans(htn(d, \phi), s, htn(d_1, \phi_1), s_1), !, \\
& \quad\ online(htn(d_1, \phi_1), s_1, s_2).
\end{aligned}
\tag{15}
$$

4 Discussions and Conclusions

Research in agent-oriented programming languages can be divided into two mainstreams. One is to develop new languages to accommodate new needs or new agent-architecture. The other is to add new features to existent language. Our work in this paper belongs to the second approach. We extended *ConGolog* by adding to it the *htn*-construct. The new language, HTN+*ConGolog*, allows us to take advantages of the HTN formalism as well as the *ConGolog* language thus allowing us to be more flexible in writing plans or control programs. Furthermore, temporal constraints can be specified easily with the new construct. HTN+*ConGolog* differs from other extensions of *Golog* such as the adding of sensing actions [9] or temporal reasoning [16] in that it does not increase the expressiveness of *ConGolog* but its flexibility.

In agent programming languages supporting the BDI-architecture such as AgentSpeak(L) [12], 3APL [6], AGENT-0 [15] etc., an agent is characterized by its mental state and a set of practical reasoning rules. These languages are rule-based languages where rules represent the know-how of the agent and are used for planning.

Informally, a rule says that if an agent wants to achieve a goal G and believes that the situation is S, then plan P might be the right thing to do. As such, to write an agent program in one of these languages means to specify its mental state, the semantics of its basic actions, and a set of practical reasoning rules. This is very similar to what a *Golog/ConGolog* programmer needs to do when writing a *Golog/ConGolog* program: he has to represent the initial situation and the action domain, and to write procedures that help achieving the goal. In both cases, the programmer does not specify exactly what need to be done to achieve the goal. The agent, while executing the program, will decide what actions need to be executed. While searching for plans are supported in AgentSpeak(L) or 3APL by having plan rules hierarchically structured, searching for plans is supported in *Golog/ConGolog* by having nondeterministic constructs. The htn-construct increases *Golog/ConGolog* flexibility by allowing hierarchical task networks to be specified easily. We believe that this is an important contribution of this paper.

There has been a lot of work on applying HTN-planning techniques in practical applications. Only in this volume, we can find the use of a HTN-planner in monitoring objectives queue of the DECAF system [2] or in developing a RETSINA Multi-Agent-System [11]. Our implementation of the htn-construct in PROLOG shows that hierarchical task networks can be solved effectively in logic programming. It also shows that the HTN-planning techniques can be useful in a high-level execution language such as *Golog/ConGolog*. In the near future, we plan to complete the integration of HTN into *ConGolog* by allowing *ConGolog* programs to occur inside htn-programs.

References

1. K. Erol, J. Hendler, and D. Nau. HTN-planning: Complexity and Expressivity. In *AAAI 94*, pages 1123–1228, 1994.
2. J.R. Graham and K.S. Decker. Towards a Distributed, Environment-Centered Agent Framework. Pre-Proceedings ATAL'99, 162–175, 1999.
3. G. De Giacomo, Y. Lespérance, and H. Levesque. Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. In *IJCAI 97*, pages 1221–1226, 1997.
4. G. De Giacomo, Y. Lespérance, and H. Levesque. *ConGolog* , a concurrent programming language based on the situation calculus: the foundation. To appear in AIJ
5. G. De Giacomo, Y. Lespérance, and H. Levesque. *ConGolog* , a concurrent programming language based on the situation calculus: language and implementation. To appear in AIJ
6. K.V. Hindriks, F.S. de Boer, W. van der Hoek, J.J. Ch. Meyer. Formal Semantics for an Abstract Agent Programming Language. *Intelligent Agents IV*, LNAI 1365, 215–229, 1998.
7. Y. Lespérance, K. Tam, and M. Jenkin. Reactivity in a Logic-Based Robot Programming Framework. In Pre-Proceedings ATAL'99, 32–46.
8. H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–84, April-June 1997.
9. G. Lakemeyer. On Sensing and Off-line Interpreting in GOLOG. *AAAI Fall Symposium, Cognitive Robotics*, 91–97, 1998.

10. J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, Edinburgh, 1969.
11. M. Paolucci, D. Kalp, A. Pannu, O. Shehory, and K. Sycara. A Planning Component for RETSINA Agents. In Pre-Proceedings ATAL'99, 235–250.
12. A.S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In W. van der Velde and J.W. Perram, editors, *Agents Breaking Away*, 42–55, 1996.
13. R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation*, pages 359–380. Academic Press, 1991.
14. E.D.Sacerdoti. Planning in a Hierarchy of Abstraction Spaces. *Artificial Intelligence*, 5 (1974), 115–135.
15. Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60 (1993), 51–92.
16. R. Reiter. Sequential, Temporal GOLOG. In KR, 1998.