Hierarchical Task Libraries in (Con)Golog

Alfredo Gabaldon
National ICT Australia School of CS & Eng.
Kensington Research Lab U. of New South Wales
Sydney, Australia
alfredo@cse.unsw.edu.au

Abstract

We are interested in building libraries of complex actions, or *tasks*, in the Situation Calculus based languages Golog and its variants. We consider simple ways of organizing tasks in a hierarchical fashion by defining generalized tasks from specialized ones. We use a military operations planning domain to illustrate these ideas.

Introduction

Complex activity planning such as that required in military operations, the construction industry, refinery maintenance, software development, and many others, is usually carried out by expert hand from scratch or by reuse and adaptation of solutions to similar, previous problems. Computer assistance is used for solving the resulting scheduling problem, usually only after the set of activities necessary to achieve the desired goal has been chosen. The activity planning phase is thus time consuming because it is done by hand, and expensive because it requires domain expertise.

We are interested in building libraries of tasks in the high-level language Golog (Levesque *et al.* 1997) or its concurrent counterpart ConGolog (De Giacomo, Lesperance, & Levesque 2000). One motivation is the possibility of mitigating the time and cost overhead in complex activity planning by developing techniques that allow the reuse of libraries of tasks in an (ideally) automatic way. Domain experts would still be required during the initial construction of such a library (no way around that). However, the idea is that once a library containing a substantial amount of knowledge in the form of domain-specific, detailed tasks for achieving a number of goals, the cost incurred in this initial phase would be amortized over the many problems solved by reusing that knowledge. In order to achieve this, the library has to be in a form that can be reused by users who are not necessarily domain experts. For example, we would like the library to be such that planning for the construction of a building on a soft clay ground will not require consulting with an expert on how to lay foundations on such a ground, if the library contains tasks for doing that. In this case, it should be enough to simply specify the type of ground, the type of building, and so on, in order to obtain from the library a detailed construction plan. And if some details are missing or underspecified, we ultimately would like the library to provide alternative plans, possibly including costs and risks.

Toward this goal, we start by considering simple ways of constructing task hierarchies. Given a set of highly specialized tasks, we consider defining new tasks that generalize the specialized ones in different ways. In defining relationships between more general and more specifics tasks, we consider imposing constraints on parameters and the use of knowledge about *types* and subtypes associated with objects of the domain.

We will explore these ideas in the context of the military operations planning domain from (Aberdeen, Thiébaux, & Zhang 2004), which we describe next.

Application Domain

Our application domain comes from a military operations planning domain described in (Aberdeen, Thiébaux, & Zhang 2004). Roughly, a domain description consists of a set of *tasks*, such as "anti-submarine operation", which are a similar but more complex than the operators or actions in classical planning. Also given is a description of the initial state, which usually consists of only a listing of what resources are available and in what quantity. The tasks are specified with a given duration, multiple possible outcomes with corresponding probabilities, resource requirements and other preconditions, and are described as capable of achieving a particular goal. Some tasks also require that other tasks be carried out prior or during their execution. These last two characteristics of tasks in this application resemble less classical planning than how Hierarchical Task Networks (HTNs) (Sacerdoti 1974; Erol, Hendler, & Nau 1996) and complex action languages like Golog operate. Hence our appeal to the latter in this work.

We are also given a goal to be achieved which may include some constraints on duration and resource consumption, and the problem consists in finding a set of tasks to achieve the goal. For instance, a goal may be "control of Island" and a task could be "anti-submarine operation with frigates".

The description of the task "anti-submarine operation with frigates" may be as follows:

- task: $antiSubOperWfrigates$

- duration: 4
- probability of success: 0.9
- preconditions: 2 frigates, air-space control
- immediate effects: -2 frigates
- effects at successful termination: seaSubSurface control established, +2 frigates
- effects at termination with failure: +1 frigate

Other tasks are more complicated, requiring the execution of other tasks prior to starting, support tasks to execute concurrently, and so on.

Querying a Task Knowledge Base

The task libraries we are interested in contain tasks that are probabilistic, use different amounts of resources such as $frigates$, and have different durations. Moreover, one of the main motivating applications is decision support. We are therefore interested in a wide variety of queries. Here are some examples in English:

- Q1: "can we achieve control of region R?"
  if R is a *land region*, yes, with prob=0.78;
  if R is a *sea region*, yes, with prob=0.85
- Q2: same as above, after adding the fact "R is of type *air space region*"
  no
- Q3: same as above, after adding the fact "12 units of *fighter aircraft* are available"
  yes, with prob=0.95
- Q4: "can we achieve control of region R by using *frigates*"
  if R is a *sea surface*, yes, with prob=...;
  if R is a *sea sub surface*, yes, with prob=...
- Q5: "can we achieve control of region R with prob. of success > 0.9?"
  if > 12 *special forces* are available, yes, with prob=...
- Q6: "can we achieve control of region R with loss of aircraft < 10?"
  if ..., yes, with prob=...
- Q7: "can we achieve control of region R in time < 4 hours?"
  if > 20 *special forces* are available, yes, with prob=...

Golog/ConGolog

Using a situation calculus axiomatization of primitive actions due to (Reiter 1991), the high-level action languages Golog (Levesque *et al.* 1997) and ConGolog (De Giacomo, Lesperance, & Levesque 2000) provide Algol-like programming constructs for defining complex actions in terms of simpler ones. We want to use (variants of) these languages to build our libraries of tasks, so in this section we review some of the constructs available in them.

The following are some of the constructs provided by Golog:

- Test condition: $\phi?$. Test whether ϕ is true in the current situation.
- Sequence: $\delta_1; \delta_2$. Execute program δ_1 followed by program δ_2.
- Nondeterministic action choice: $\delta_1 | \delta_2$. Nondeterministically choose between executing δ_1 or executing δ_2.
- Nondeterministic choice of arguments: $(\pi x)\delta$. Choose a value for x and execute δ with such a value for x.
- Procedure definitions: **proc** $P(\vec{x})$ δ **endProc** . $P(\vec{x})$ being the name of the procedure, $\vec{x}$ its parameters, and program δ its body.

ConGolog extends Golog with a number of constructs. The most relevant for our purposes is the following:

- concurrent execution: $\delta_1 \parallel \delta_2$

Intuitively, the execution of a complex action $\delta_1 \| \delta_2$ results in the execution of one of the possible interleavings of the primitive actions that result from δ_1 with those from δ_2. If there is no interleaving that is executable (the preconditions of each action are satisfied at the appropriate moment), then the execution of $\delta_1 \| \delta_2$ fails.

Representing Tasks

We will describe a task tsk by a list of statements of the following forms:

- $task(tsk, x_1, \ldots, x_n, \delta)$, meaning that tsk is a task with parameters $x_1, \ldots, x_n$ and the Golog program δ as its body. Variables $x_1, \ldots, x_n$ may appear free in δ;
- $tskTypes(tsk, t_1, \ldots, t_n)$, meaning that the parameters of tsk are of types $t_1, \ldots, t_n$;
- $tskRes(tsk, res, resn)$, meaning that tsk requires amount $resn$ of resource res;
- $tskDur(tsk, dur)$, meaning that tsk has duration dur;
- $achievesG(tsk, \phi)$, meaning that task tsk is capable of achieving goal ϕ.

For example, an "anti-sub operation with submarines" task might be specified by means of the list of statements:

$$\begin{array}{l} task(aswWsubs, r, t, \delta_1), \\ tskTypes(aswWsubs, seaSubSurf), \\ tskRes(aswWsubs, submarine, 2), \\ tskDur(aswWsubs, 32), \\ achievesG(aswWsubs, seaSubSurfCtrlEst). \end{array} \quad (1)$$

specifying that the task $aswWsubs$ takes two parameters: r of type $SeaSubSurf$, the special temporal argument t, and the program δ_1, which we describe below, as its body. The task requires 2 units of resource $submarine$, and has a duration of 32 units of time. The last statement indicates that this task, when successful, achieves the goal of establishing control of the region r. We discuss "achieving" goals further below.

Situation Calculus encoding

While basic action theories formalize actions that are deterministic and instantaneous, we want to formalize tasks that have a duration, may be stochastic, and may involve the execution of other tasks. We will therefore turn to the complex action languages Golog and ConGolog to encode such tasks as programs. The temporal aspect of tasks will be handled by using the technique of representing durative actions in the situation calculus as *processes*, that is, by means of a "start" and an "end" action, plus a fluent that holds in situations when the process is executing.

Consider the task "anti-sub operation with submarines" mentioned above. This task takes a time t and a region r as parameters. The region r must be of type "sea sub-surface" (denoted by $SeaSubSurf$). The main effect of the task, when successfully executed, is "control established" for the region r. There may be other effects such as resource consumption. Suppose we specify a duration of 32 units of time.[1] The task is defined by means of the statements (1), where δ_1 stands for the following Golog program:

$$\delta_1 = \left\{ \begin{array}{l} (\pi\ t_e).endTime(aswWsubs, t, t_e)\ ; \\ st\_aswWsubs(r, t)\ ; \\ end\_aswWsubs(r, t_e) \end{array} \right.$$

Here, the expression $(\pi\ t_e).endTime(aswWsubs, t, t_e)$ is used to compute the time the task is to terminate. $endTime$ is a macro defined as follows:

$$endTime(tsk, t, t_e) \stackrel{\text{def}}{=} (\pi d)[tskDur(tsk, d)?\ ;\ (t_e = t + d)?].$$

The last two lines in δ_1 are actions for starting and terminating the task. The start action, $st\_aswWsubs(r, t)$ is a stochastic action (and modeled as in stGolog (Reiter 2001)). The terminating action $end\_aswWsubs(r, t)$ is a primitive deterministic action. In our military operations planning example, we only consider two outcomes for stochastic actions: success and failure. For action $st\_aswWsubs(r, t)$, the outcomes are modeled by means of the primitive actions $stS\_aswWsubs(r, t)$ and $stF\_aswWsubs(r, t)$. In general, however, any finite number of outcomes is possible.[2] These actions are axiomatized as follows.

- Action precondition axioms for the primitive start and end actions:

$$\begin{array}{l} Poss(stS\_aswWsubs(r, t), s) \equiv \\ \quad (\exists s)available(submarines, n, s) \wedge n >= 2, \\ Poss(stF\_aswWsubs(r, t), s) \equiv \\ \quad (\exists s)available(submarines, n, s) \wedge n >= 2, \\ Poss(end\_aswWsubs(r, t), s) \equiv aswWsubsExe(r, s). \end{array}$$

  The only requirement for starting this task, is that two submarines be available. Terminating the task is possible iff the task is executing.

- A "process" fluent that holds while the task is occurring:

$$\begin{array}{l} aswWsubsExe(r, do(a, s)) \equiv \\ \quad (\exists t).a = stS\_aswWsubs(r, t) \vee \\ \quad\quad a = stF\_aswWsubs(r, t) \vee \\ \quad aswWsubsExe(r, s) \wedge \\ \quad\quad \neg(\exists t)a = end\_aswWsubs(r, t). \end{array}$$

- Choice of outcomes and their probabilities for the stochastic action $st\_aswWsubs(r, t)$:

$$\begin{array}{c} choice(st\_aswWsubs(r, t), c) \equiv \\ c = stS\_aswWsubs(r, t) \vee c = stF\_aswWsubs(r, t), \end{array}$$

$$\begin{array}{l} prob_0(stS\_aswWsubs(r, t), st\_aswWsubs(r, t)) = 0.8 \\ prob_0(stF\_aswWsubs(r, t), st\_aswWsubs(r, t)) = 0.2 \end{array}$$

Let us consider next a task that triggers the execution of a subtask before it starts. Similar to the task above, it represents an anti-submarine operation that achieves control of a region of type "sea sub-surface," but uses frigates instead of submarines. Because it uses frigates, it requires control of the air-space before it starts. This task, with a given duration of 4 units of time, is defined as follows:

$$\begin{array}{l} task(aswWfrigates, r, t, \delta_2), \\ tskTypes(aswWfrigates, SeaSubSurf), \\ tskRes(aswWfrigates, frigate, 2), \\ tskDur(aswWfrigates, 32), \\ achievesG(aswWfrigates, controlEst(r)), \end{array}$$

where

$$\delta_2 = \left\{ \begin{array}{l} achieve(airCtrlEst, r, t)\ ; \\ (\pi\ tnow).start(now, tnow)?\ ; \\ (\pi\ t_e).endTime(aswWfrigates, tnow, t_e)?\ ; \\ st\_aswWfrigates(r, tnow)\ ; \\ end\_aswWfrigates(r, t_e) \end{array} \right.$$

The first line calls a procedure $achieve(airCtrlEst, r, t)$. By means of this special procedure, tasks may invoke other tasks in order to satisfy some condition (in this case, air control of region r). Note that the task does not invoke any particular task to satisfy the condition. The achieve procedure will, non-deterministically, choose a task among those specified as capable of achieving the goal "air control established," and execute it. The information about which tasks are capable of achieving which goals is provided by the knowledge engineer by means of $achievesG$ statements as shown earlier. In a later section we show how the $achieve$ procedure is defined. Once such a subtask has executed, the current time $tnow$ is obtained, and start and end actions are subsequently executed at the appropriate times.

Since the attempt to achieve "air control established" could have failed, the action $st\_aswWfrigates(r, t)$ has that property as a precondition, in addition to the required number of resources:

$$\begin{array}{l} Poss(st\_aswWfrigates(r, t), s) \equiv \\ \quad airCtrlEst(r, s) \wedge \\ \quad available(frigates, 2, s). \end{array}$$

[1] For simplicity, we use constant amounts of resources and durations, but in general these may be determined by other factors, such as properties of the current situation.

[2] The probabilistic part of the formalization is important but not our main concern in this paper, thus from now on we leave it out.

In general, tasks will be ConGolog programs of the form:

$$
\begin{array}{l}
[achieve(subtask_1, \vec{x}_1, t) \parallel \\
\ldots \\
achieve(subtask_k, \vec{x}_k, t)] \ ; \\
(\pi\, tnow).start(now, tnow)? \ ; \\
(\pi t_e).endTime(tsk, tnow, t_e)? \ ; \\
st\_task(\vec{x}, tnow) \ ; \\
end\_task(\vec{x}, t_e)
\end{array}
$$

We use programs with the ConGolog construct for concurrency, $\parallel$, because the conditions required to be satisfied before the task starts can often be attempted in parallel. If some need to be attempted earlier than others, the $\parallel$ construct can be replaced with Golog's sequential construct ; and nesting used as necessary.

Task Execution

A task is executed by 1) checking the arguments are of the right types, 2) substituting the arguments in the body of the task, and 3) executing the body. Let us introduce two more abbreviations. One for type checking the arguments of a task:

$$
\begin{array}{c}
typeCheck(tsk, x_1, \ldots, x_n) \stackrel{\text{def}}{=} \\
(\exists t_1, \ldots, t_n)[tskTypes(tsk, t_1, \ldots, t_n) \wedge \\
\bigwedge_{i=1,\ldots,n} type(x_i, t_i)]
\end{array}
$$

and the other one for task execution:

$$
\begin{array}{l}
execTask(tsk, \vec{x}) \stackrel{\text{def}}{=} \\
\quad typeCheck(tsk, \vec{x})? \ ; \\
\quad (\exists \delta).task(tsk, \vec{x}, \delta)? \ ; \\
\quad \delta
\end{array}
\qquad (2)
$$

So executing, say, task $aswWsubs$ with arguments $Region1$ and time T_0, is implemented as the execution of the (Con)Golog program:

$$execTask(aswWsubs, Region1, T_0).$$

We are assuming here that type information for domain objects is contained in the knowledge base through a predicate $type(x, t)$. We briefly come back to this later.

Achieving Goals

The two tasks discussed above, $aswWsubs$ and $aswWfrigates$, represent two different ways of achieving the same thing, namely, establishing control of a region of type "sea sub-surface". They differ on the means used to achieve this goal: one uses submarines and the other uses frigates, and have different durations, success probabilities, and subtask requirements.

Using these two tasks, we specify an *achieve* procedure for the goal "establish sea subsurface control", denoted by the fluent $seaSubSurfCtrl$. This procedure will non-deterministically choose among the tasks specified as capable of achieving this goal and execute it. This simple procedure is defined in Golog as follows:

proc $achieve(seaSubSurfCtrl, reg, t)$
  $seaSubSurfCtrl(reg)? \mid$
  $execTask(aswWsubs, reg, t) \mid$
  $execTask(aswWfrigates, reg, t)$
endProc

Notice that one of the choices for achieving this goal is to do nothing other than successfully test that the goal is already true. The other two choices consist in executing one of the tasks discussed earlier, that were specified by an $achievesG$ statement as capable of achieving this goal.

Generalized Tasks

A collection of tasks can also be organized based on the level of generality. A task can be related to another task by being a more general or specific version of another. A task may be more specific than another by virtue of taking a subset of the parameters, or because it requires some of the parameters to have a specific value, or because it requires some of the parameters to be of a subtype of the type of parameters accepted by the more general task.

For example, suppose that in addition to the tasks $aswWsubs$ and $aswWfrigates$, which take a region of type $seaSubSurf$ (sea sub-surface) as parameter, there is also a task $marEscortOp$ (maritime escort operation) which has a region of type $seaSurf$ (sea surface) as a parameter and is capable of producing $seaSurfCtrl$ as an effect (it establishes control of the sea surface region). Suppose also that we know that $seaSurf$ and $seaSubSurf$ are both subtypes of $seaRegion$. Then, given the definitions of these three tasks, we may want to define another task, generalizing them, that takes a parameter of type $seaRegion$ and, when successful, establishes control of such a region. A simple approach one can follow then is to define the general task as the execution of one of the more specialized tasks, according to the subtype of the parameter passed to it. Since type checking is done as part of task execution (see abbreviation (2)), we can define the general task using the non-deterministic choice construct of Golog:

$$
\begin{array}{l}
execTask(ctrlSeaRegion, reg, t) \stackrel{\text{def}}{=} \\
\quad execTask(aswWsubs, reg, t) \mid \\
\quad execTask(aswWfrigates, reg, t) \mid \\
\quad execTask(marEscortOp, reg, t).
\end{array}
$$

For the case where tasks are more specific by virtue of assuming some of the parameters to have specific values, we need to include these constraints on parameters. Consider an example borrowed from (Lifschitz & Ren 2006) (this proceedings) where a general *move* action is related to the specialized move actions *push box* and *walk*. The push box action is always performed by $Monkey$ and the object being moved is always Box. Thus the action has only one parameter, loc, for the location the box is pushed to. The action $walk$ takes two parameters: $agent$ and loc, for the agent doing the walking and the location the agent is walking to. The definitions of these tasks may include the following assertions: $task(pushBox, loc, t, \delta_{pB})$ and

$task(walk, agent, loc, t, \delta_w)$. A generalized move action could be defined in terms of these specialized actions, by including constraints on parameters together with execution invocation:

$$execTask(move, a, o, l, t) \stackrel{\text{def}}{=}$$
$$[(a = Monkey \wedge o = Box)?\ ;\ execTask(pushBox, l, t)]\ |$$
$$[(a = o)?\ ;\ execTask(walk, a, l, t)].$$

It is easy to imagine a huge number of specialized versions of $move$ that could be included in the above definition. For example, driving, taking a bus, riding a bike, would be variations of move similar to walking in that the object being moved is the agent itself.

On the other hand, we may also want to consider cases where the more specific task requires more, rather than less, parameters than the general one. For instance, riding a bus from some location to another, could have a bus route number and a pair of locations (origin/destination) as parameters. In this case defining riding a bus as a special case of move requires choosing values for the extra parameters:

$$execTask(move, a, o, l, t) \stackrel{\text{def}}{=}$$
$$(a = o)?\ ;$$
$$(\pi\ bus,\ stop)$$
$$[(\exists l')(atLoc(a, l') \wedge nearby(l', stop))?\ ;$$
$$execTask(rideBus, a, bus, stop, l, t)]\ |$$
$$\ldots$$

Type Information

As we mentioned earlier, part of the information that we assume to be encoded together with the task library is type information. We are assuming that the knowledge base includes information such as $type(Region1, seaSubSurf)$.

Generalized tasks together with type information allows for interesting queries, such as Q1, where we know R is of type *region* but have no information on whether it is a *land region*, *sea region*, or a *airspace region*. Given a task that generalizes tasks specific for these three regions, we may compute a corresponding answer for each of the three types of region. We are currently working on a suitable treatment of type information, which appears to be necessary in order to handle the more interesting queries.

Implementation

We have implemented a version of the task library for the military operations planning domain. Building on the interpreters of some of the variations of Golog and ConGolog available from the U. of Toronto Cognitive Robotics group, we put together an interpreter for a stochastic, temporal version of ConGolog (let us call it sttConGolog). The interpreter is written in Eclipse Prolog and it uses Eclipse's constraint library to solve the constraint problem that arises from the use of rationals as the temporal domain in our sttConGolog programs.

We adapted part of the stochastic Golog (stGolog) interpreter (Reiter 2001) that allows us to compute the probability of a fluent formula being true after executing a sttConGolog program. This allows us to compute answers to queries of the forms Q1–Q3, Q5, and Q7. Currently we are not able to answer queries such as Q4 and Q6 that include constraints on resources such as *frigates* and *aircraft*. This will require a more explicit treatment of resources, but we believe that it will not be too difficult to add.

Conclusions

In this paper we discuss our very preliminary steps towards building a library of tasks that would be incorporated into a decision support knowledge base. We have used a military operations planning domain to illustrate how stochastic, temporal tasks can be encoded in the complex action languages Golog/ConGolog, and organized into a hierarchy according to specificity of the tasks and of type information. Such a hierarchical organization of a task library seems to be useful in the development of a query answering system.

References

Aberdeen, D.; Thiébaux, S.; and Zhang, L. 2004. Decision-theoretic military operations planning. In *Procs. of the 14th International Conference on (ICAPS'04)*.

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

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.; 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.

Lifschitz, V., and Ren, W. 2006. Towards a modular action description language. In *Procs. of the AAAI Spring Symposium: Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering*.

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.