1

Programming Hierarchical Task Networks in the Situation Calculus

ALFREDO GABALDON

ABSTRACT. Hierarchical Task Networks (HTNs) is a successful approach for planning where domain specific knowledge is used in order to make the search for plans more efficient. Similarly, high-level action programming languages like Golog provide a way to use procedural domain knowledge as a search control mechanism in planning. In this work, we present a systematic encoding of HTNs into Golog/ConGolog, allowing one to use both forms of search control under the same framework and also shedding some light on the relationship between the two formalisms.

Foreword

Some years ago, on Hector's suggestion[1] a research effort was begun that has resulted in the Golog family of action programming languages and the new field of *Cognitive Robotics*. It also led to the creation of the Cognitive Robotics Group where as a student I had the fortune of interacting with and learning from Hector. I am pleased to make a contribution to this collection in honor of Hector's 60th birthday.

1 Introduction

Recognizing the intractability of domain-independent planning, researchers have proposed various approaches to take advantage of frequently available domain specific knowledge, to speed up the search for a plan. One of these approaches is Hierarchical Task Network planning, which originated in the work of Sacerdoti [Sacerdoti 1974] and Tate [Tate 1977]. This approach, to which we will refer to as HTN-planning, computes a set of primitive actions or operators just as in (classical) planning. However, instead of a goal condition to be achieved by the actions, in HTN-planning problem is described as a set of *tasks* to be performed. The domain specific knowledge is provided in the form of task decomposition directives, that is, the HTN-planning system 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 is solved when a sequence of primitive tasks is found that corresponds to performing the original set of tasks.

The intractability of classical planning, in particular for high-level robotic control, was one of the motivations [Levesque and Reiter 1998] behind the introduction of the action programming language Golog [Levesque, Reiter, Lespérance, Lin, and Scherl 1997]. In this approach, instead of relying on classical planning, high-level control is driven by programs written in a language with procedural constructs like conditionals, loops and procedure definitions and calls, and whose primitive statements are *actions* from an underlying action

[1] as recounted in [Reiter 2001].

theory. Although there is more to these programs, one way to see them is as providing a means to give the robot a more specific idea of how to do what it needs to do.

Yet another approach was advanced by Bacchus and Kabanza [2000] in their TLPlan planning system. Their approach consists in providing a planning system with domain specific knowledge in the form of Linear Temporal Logic formulae, to help the system discard early some of the unpromising partial plans. Here we will not consider this approach, but see Section 5 for some comments on related work.

Our aim here is to look at the relationship between HTN-planning and high-level action programming languages such as Golog. More specifically, we present an encoding of HTN-planning problems in Golog and ConGolog [De Giacomo, Lesperance, and Levesque 2000]—an extension of Golog with concurrency and interrupts. In addition to the formal result itself showing that HTN-planning can be embedded into Golog, there are several practical benefits from the embedding. One obvious benefit is that instead of having to manually re-encode knowledge given in the form of an HTN, one can use the systematic encoding introduced here to incorporate or mix it with procedural knowledge given in the form of Golog programs. Second, the encoding brings to HTN-planning a considerable set of new features which have been formalized and are available in the Golog family of languages. For instance: explicit time, sensing actions, exogenous events, execution monitoring, incomplete information about the initial state and stochastic actions, to mention a few, all within a single, coherent logical framework. Thus the range of problems that can be tackled potentially becomes much larger. As an illustration of this, we will use the ConGolog encoding of a logistics domain HTN, and add to it run-time package delivery requests (exogenous actions).

2 Preliminaries

2.1 The Situation Calculus

The Situation Calculus [McCarthy 1963] is a logical language for describing and reasoning about dynamic worlds. It has three main components which are used to talk about change in a dynamic world: actions, which are assumed to be the cause of all change in the world and are treated as first-class objects in the language; situations, which correspond to possible states the world may evolve into and are also treated as fist-class objects; and fluents, relations or functions representing the properties of the world that change as the world evolves due to the occurrence of actions.

In this work we use the particular axiomatization of the Situation Calculus that has been developed by Levesque, Reiter and their colleagues and students in the Cognitive Robotics group at the U. of Toronto [Reiter 1991; Levesque, Pirri, and Reiter 1998; Reiter 2001]. This axiomatization uses a classical logic language with sorts $action$, $situation$ and $object$. A special constant S_0 is used to denote the initial situation, and the function symbol do of sort $(action, situation) \mapsto situation$ is used to represent situations that result from executing actions, that is, a term $do(\alpha, \sigma)$, where α is an action term and σ a situation term, represents the situation that results from doing α in situation σ. By nesting function do, it is possible to build sequences of actions. For instance, a sequence consisting of actions a_1, a_2, a_3 is represented by the term $do(a_3, do(a_2, do(a_1, S_0)))$.

Fluents are represented by means of relations $F(\vec{x}, s)$ where $\vec{x}$ is a tuple of arguments of sorts $object$ or $action$ and the last argument, s, is always of sort $situation$. For example, a fluent $atTruck(trk, loc, s)$ could be used to represent the location loc of a tuck trk in

situation s.

Function symbols of sort $object \mapsto action$ are used for terms $A(\vec{x})$ that represent action types. For instance, a function *loadTruck*(obj, trk) could be used to represent the action of loading an object obj onto a truck trk. They are called action *types* because a single function symbol can be used to create multiple *instances* of an action, e.g. the instances *loadTruck*(Box_1, Trk_1), *loadTruck*(Box_2, Trk_2), etc. We will use $a_1, a_2, \ldots$ to denote action variables, $\alpha_1, \alpha_2, \ldots$ to denote action terms, and $s_1, s_2, \ldots$ to denote situation variables.

A Basic Action Theory $\mathcal{D}$ consists of the following sets of axioms (variables that appear free are implicitly universally quantified. $\vec{x}$ denotes a tuple of variables $x_1, \ldots, x_n$):

1. For each action type $A(\vec{x})$, there is exactly one **Action Precondition Axiom** (APA) of the form:

$$Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)$$

   where variable s is the only term of sort situation in formula $\Pi_A(\vec{x}, s)$. The latter formula represents the conditions under which an action $A(\vec{x})$ is executable. The restriction that the only situation term mentioned in this formula be s intuitively means that these preconditions depend only on the situation where the action would be executed.

2. For each fluent $F(\vec{x}, s)$, there is exactly one **Successor State Axiom** (SSA) of the form:

$$F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s)$$

   where s is the only term of sort situation in formula $\Phi_F(\vec{x}, a, s)$. This formula represents all and the only conditions under which executing an action a in a situation s causes the fluent to hold in situation $do(a, s)$. These axioms embody Reiter's solution to the frame problem [Reiter 1991; Reiter 2001].

3. The **Initial Database** $\mathcal{D}_{S_0}$ which is a finite set of sentences whose only situation term is S_0 and describe the initial state of the domain, i.e. before any actions have occurred. Any sentence is allowed as long as the only situation term that appears in it is S_0, so one can write sentences such as $(\exists box) atLoc(box, L, S_0)$, reflecting incomplete information about the initial state of a domain.

4. The **Foundational Axioms** Σ which define situations in terms of the constant S_0 and the function do. Intuitively, these axioms define a tree-like structure for situations with S_0 as the root of the tree. They also define relation $\sqsubset$ on situations. Intuitively, $s \sqsubset s'$ means that the sequence of actions s is a prefix of sequence s'.

5. A set of unique names axioms (UNA) for actions. For example,

$$\begin{aligned} &\textit{loadTruck}(o, t) \neq \textit{driveTruck}(t, l_1, l_2), \\ &\textit{loadTruck}(o, t) = \textit{loadTruck}(o', t') \supset (o = o' \wedge t = t') \end{aligned}$$

EXAMPLE 1. Through out this chapter we will use the well known logistics domain as an example. In this domain, there are objects that need 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 plane, driving a truck and flying a plane. A basic action theory for this domain is comprised of a set of axioms that includes the following:

Action Precondition Axioms:

$$Poss(\mathit{loadTruck}(o, tr), s) \equiv (\exists l).atTruck(tr, l, s) \wedge atObj(o, l, s),$$

$$Poss(\mathit{unloadTruck}(o, tr), s) \equiv inTruck(o, tr, s),$$

$$Poss(loadAirplane(o, p), s) \equiv (\exists l).atObj(o, l, s) \wedge atAirplane(p, l, s),$$

$$Poss(unloadAirplane(o, p), s) \equiv inAirplane(o, p, s),$$

$$Poss(driveTruck(tr, or, de), s) \equiv atTruck(tr, or, s) \wedge (\exists c).inCity(or, c) \wedge inCity(de, c).$$

$$Poss(fly(p, or, de), s) \equiv atAirplane(p, or, s) \wedge airport(de).$$

Successor State Axioms:

$$\begin{aligned} atObj(o, l, do(a, s)) \equiv\ & (\exists tr)[a = unloadTruck(o, tr) \wedge atTruck(tr, l, s)] \vee \\ & (\exists p)[a = unloadAirplane(o, p) \wedge atAirplane(p, l, s)] \vee \\ & atObj(o, l, s) \wedge \neg(\exists tr)a = \mathit{loadTruck}(o, tr) \wedge \\ & \neg(\exists p)a = loadAirplane(o, p). \end{aligned}$$

$$\begin{aligned} atTruck(tr, l, do(a, s)) \equiv\ & (\exists or)a = driveTruck(tr, or, l) \vee \\ & atTruck(tr, l, s) \wedge \neg(\exists or, de)a = driveTruck(tr, or, de), \end{aligned}$$

$$\begin{aligned} atAirplane(p, ap, do(a, s)) \equiv\ & (\exists oap)a = fly(p, oap, ap) \vee \\ & atAirplane(p, ap, s) \wedge \neg(\exists oap, dap)a = fly(p, oap, dap), \end{aligned}$$

$$\begin{aligned} inTruck(o, tr, do(a, s)) \equiv\ & a = \mathit{loadTruck}(o, tr) \vee \\ & inTruck(o, tr, s) \wedge a \neq unloadTruck(o, tr), \end{aligned}$$

$$\begin{aligned} inAirplane(o, p, do(a, s)) \equiv\ & a = loadAirplane(o, p) \vee \\ & inAirplane(o, p, s) \wedge a \neq unloadAirplane(o, p). \end{aligned}$$

Initial situation. These would include sentences such as:

$$atAirplane(p, l, S_0) \equiv p = Plane_1 \wedge l = Loc_{5,1} \vee p = Plane_2 \wedge l = Loc_{2,1}.$$

$$atTruck(t, l, S_0) \equiv t = Truck_{1,1} \wedge l = Loc_{1,1} \vee t = Truck_{2,1} \wedge l = Loc_{2,1} \vee \ldots$$

$$airport(loc) \equiv loc = Loc_{1,1} \vee loc = Loc_{2,1} \vee loc = Loc_{3,1} \vee \ldots$$

$$inCity(l, c) \equiv l = Loc_{1,1} \wedge c = City_1 \vee l = Loc_{2,1} \wedge c = City_2 \vee \ldots$$

$$atObj(p, l, S_0) \equiv p = Package_1 \wedge l = Loc_{3,3} \vee p = Package_2 \wedge l = Loc_{3,1} \vee \ldots$$

The above set of axioms is almost a complete basic action theory for the logistics domain. The only axioms missing are the domain independent foundational axioms and the unique names axioms for actions. Note that the initial situation includes non-fluent relations such as $airport(loc)$ and $inCity(l, c)$. It may also include other "utility" sentences such as formulae specifying that different constants denote different objects, e.g. $Loc_{1,1} \neq Loc_{1,2}$.

Basic action theories in the Situation Calculus is one of the established formalisms for reasoning about actions. It has been used to formalize various reasoning problems in terms of logical deduction, including the classical planning problem: given a basic action theory, $\mathcal{D}$, describing the planning domain and including a description of the initial state, and given also a goal formula $G(s)$, the planning problem is specified in terms of logical entailment as follows:

$$\mathcal{D} \models (\exists s).executable(s) \wedge G(s)$$

where $executable(s)$ intuitively means that the situation s, i.e. the plan, includes only actions whose preconditions are satisfied.

2.2 Golog and ConGolog

The high-level action programming languages Golog [Levesque, Reiter, Lespérance, Lin, and Scherl 1997] and ConGolog [De Giacomo, Lesperance, and Levesque 2000] are defined on top of a basic action theory and provide a set of programming constructs that allow one to define compound actions in terms of simpler ones. These constructs are the typical constructs of an imperative programming language. Golog includes the following:

- Test condition: $\phi?$. Test whether ϕ is true in the current situation.
- Sequence: $\delta_1; \delta_2$. Execute δ_1 followed by δ_2.
- Non-deterministic action choice: $\delta_1 | \delta_2$. Execute δ_1 or δ_2.
- Non-deterministic choice of arguments: $(\pi x)\delta$. Choose a value for x and execute δ for that value.
- Non-deterministic iteration: δ^*. Execute δ zero or more times.
- Procedure definitions: **proc** $P(\vec{x})$ δ **endProc** . $P(\vec{x})$ is the name of the procedure, $\vec{x}$ its parameters, and δ is the body.

ConGolog includes the above constructs plus the following:

- Synchronized conditional: **if** ϕ **then** δ_1 **else** δ_2.
- Synchronized loop: **while** ϕ **do** δ.
- Concurrent execution: $\delta_1 \parallel \delta_2$.
- Prioritized concurrency: $\delta_1 \rangle\rangle \delta_2$. Execute δ_1 and δ_2 concurrently but δ_2 executes only when δ_1 is blocked or done.
- Concurrent iteration: $\delta^{\parallel}$. Execute δ zero or more times in parallel.
- Interrupt: $\phi \rightarrow \delta$. Execute δ whenever condition ϕ is true.

EXAMPLE 2. In the logistics domain, moving an object to a new location may be achieved by transporting it within a city, or it may require flying it to another city. So it may be useful to define a complex action $moveObj(o, loc)$ that invokes the required actions to move o to the new location loc.

Such a complex action can be defined as a Golog procedure as follows:

proc $moveObj(o, loc)$
  $(\pi\ oloc, ocity).$
    $[atObj(o, oloc) \wedge inCity(oloc, ocity)]?$;
    if $inCity(loc, ocity)$ **then**
      $inCityDeliver(o, oloc, loc)$
    else
      $(\pi\ dcity).$
        $inCity(loc, dcity)?$;
        $(\pi\ oap, dap).$
          $[inCity(oap, ocity) \wedge inCity(dap, dcity)]?$;
          $inCityDeliver(o, oloc, oaprt)$;
          $airDeliver(o, oaprt, daprt)$;
          $inCityDeliver(o, daprt, loc)$
endProc

This procedure calls procedures $inCityDeliver(\cdots)$ and $airDeliver(\cdots)$ according to the origin and destination of the object being moved.

ConGolog is clearly a more complex language than Golog, and that is fairly evident in the logical formalization of the semantics. In fact, the semantics of Golog is defined through a set of 'macros' that stand for standard Situation Calculus formulae [Levesque, Reiter, Lespérance, Lin, and Scherl 1997; Reiter 2001]. However, since we will need the features of ConGolog to encode HTNs with partially ordered tasks and ConGolog includes all the constructs of Golog, we will only give an overview of the ConGolog semantics.

The formal semantics of ConGolog is defined in terms of a relation $Trans(\delta, s, \delta', s')$ that defines single computation steps of a program. Intuitively, the relation defines a single step transition from a *configuration* δ, s into a configuration δ', s' that results from executing one step of program δ in situation s. An additional relation $Final(\delta, s)$ defines those configurations that are terminating, i.e. where the computation may end successfully.

The full axiomatization of $Trans(\delta, s, \delta', s')$ and $Final(\delta, s)$ can be found in [De Giacomo, Lesperance, and Levesque 2000]. Below we show a sample of the axioms, which provide some idea of how it captures the semantics of each construct.

$$Trans(nil, s, \delta', s') \equiv False,$$
$$Trans(a, s, \delta', s') \equiv Poss(a, s) \wedge \delta' = nil \wedge s' = do(a, s),$$
$$Trans(\phi?, s, \delta', s') \equiv \phi[s] \wedge \delta' = nil \wedge s' = s,$$
$$Trans(\delta_1; \delta_2, s, \delta', s') \equiv$$
$$(\exists\gamma)\delta' = (\gamma; \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \ \vee Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta', s'),$$

$$Trans((\pi v)\delta, s, \delta', s') \equiv (\exists x)Trans(\delta_x^v, s, \delta', s'),$$
$$Trans(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, s, \delta', s') \equiv$$
$$\phi[s] \wedge Trans(\delta_1, s, \delta', s') \ \vee \neg\phi[s] \wedge Trans(\delta_2, s, \delta', s'),$$
$$Trans(\textbf{while } \phi \textbf{ do } \delta, s, \delta', s') \equiv$$
$$(\exists\gamma).(\delta' = \gamma \ ; \ \textbf{while } \phi \textbf{ do } \delta) \ \wedge \phi[s] \wedge Trans(\delta, s, \gamma, s'),$$
$$Trans(\delta_1 \parallel \delta_2, s, \delta', s') \equiv$$
$$(\exists\gamma)[\delta' = (\gamma \parallel \delta_2) \wedge Trans(\delta_1, s, \gamma, s')] \ \vee (\exists\gamma)[\delta' = (\delta_1 \parallel \gamma) \wedge Trans(\delta_2, s, \gamma, s')].$$

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

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

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

The axiom defining $Trans^*$, as well as one of the foundational axioms which recursively defines sequences of actions, requires second order quantification (for details see [De Giacomo, Lesperance, and Levesque 2000]). The intention with these axiomatizations is not to use them directly in implementations. The are meant to semantically characterize dynamic wolds and programs. Nevertheless, under suitable assumptions, various practical interpreters have been devised. In Section 4 we shall use on of these interpreters to demonstrate some sample runs with our HTN encoding in ConGolog.

2.3 HTN Planning

In this section we give a brief overview of HTN-planning. Our discussion is based on the definitions of HTN-planning from [Erol, Hendler, and Nau 1996], where an *operational semantics* of HTN-planning is given. HTNs are defined over a first-order language with a vocabulary consisting of sets symbols for variables, constants, predicates, primitive tasks,

compound tasks, and some other symbols. Here we will take the language from a basic action theory and just add symbols for compound tasks. The only assumption we need to make is that the set of constants is finite and denote the elements of the domain. We will then talk about ground instances of tasks with the usual meaning. Furthermore, instead of new symbols for primitive tasks, we will use action terms from an underlying basic action theory as the primitive task symbols. Finally, we use situations instead of state symbols.

A *primitive task* is an action term $A(\vec{x})$. A *compound task* is a term of the form $tname(\vec{x})$ where $tname$ is a compound task (function) symbol. A *task network* is a pair (T, ϕ) where T is a list of tasks and ϕ 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. A task network consisting only of primitive tasks is called a *primitive task network*. 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 set of methods plus a basic action theory (which includes an initial database $\mathcal{D}_{S_0}$). The parameter s is the the situation from where planning starts and in general it can be any situation. If a situation different from the initial situation, S_0, is given, then the problem involves planning after some actions have already occurred. Nevertheless, for brevity, we will only consider problems where the situation parameter is S_0. A *plan* is a sequence of ground primitive tasks.

Let d be a primitive task network, s a situation, and D a planning domain. A sequence of primitive tasks σ is a *completion* of d in s, denoted by $\sigma \in comp(d, s, D)$, if σ is a total ordering of a ground instance of the primitive task network d, it is executable in s and satisfies the constraint formula in d. For d containing a non-primitive task, $comp(d, s, D)$ is defined to be the empty set.

Let d be a task network that contains a compound task t and $m = (h, d')$ be a method such that θ 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 the tasks $t_1\theta, \ldots, t_k\theta$ from d' and incorporating (see [Erol, Hendler, and Nau 1996] for details) the constraints in d' with those in d. Define $red(d, D)$ as the set of all reductions of d by methods of D, that is $red(d, D) = \{d' \mid t \text{ is a task in } d,\ m \text{ is a method in } d \text{ and } d' = reduce(d, t, m)\}$.

A solution, $sol(d, s, D)$, of a planning problem (d, s, D) is defined recursively as follows:

$$
\begin{aligned}
&sol_1(d, s, D) = comp(d, s, D)\\
&sol_{n+1}(d, s, D) = sol_n(d, s, D) \cup \textstyle\bigcup_{d' \in red(d,D)} sol_n(d', s, D)\\
&sol(d, s, D) = \textstyle\bigcup_{n<\omega} sol_n(d, s, D)
\end{aligned}
$$

The set $sol(d, s, D)$ contains all plans that can be derived in a finite number of steps.

EXAMPLE 3. The following are methods for a task $moveObj(o, loc)$ for moving an object in the logistics domain as described in Example 2. The first method works for moving an object within the same city. The second is for moving an object between cities.[2]

[2]For improved readability, we use the labelling notation $t_i = tname(\vec{x})$ and labels t_i to refer to tasks.

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

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

3 Programming HTNs in Golog/ConGolog

In this section we show how HTN-planning problems can be encoded in Golog/ConGolog. We will considered two versions of the HTN-planning: totally ordered and partially ordered. In totally ordered HTNs, the task networks of all methods must be totally ordered. Considering totally ordered and partially ordered HTNs separately is interesting because they have different computational complexity which will indeed allow us to use the simpler language Golog to encode totally ordered HTNs. Moreover, implemented planning systems exist for each type of problem, namely, systems SHOP [Nau, Cao, Lotem, and Munoz-Avila 1999] for totally ordered HTNs and SHOP2 [Nau, Munoz-Avila, Cao, Lotem, and Mitchell 2001] for partially ordered HTNs. These systems have been very successfully applied to real world domains. Also, it is probably useful in practice to have separate solutions for each class of problem.

3.1 Totally ordered task networks

Let us then first consider totally ordered HTNs. A task network (T, ϕ) is totally ordered if the boolean formula ϕ includes precedence constraints $(t_1 \prec t_2),\ (t_2 \prec t_3), \ldots, (t_{n-1} \prec t_n)$ on the tasks in T. We will further assume that ϕ is a conjunction of the precedence constraints $t_i \prec t_j$ and of a set of constraints of the form (l, t) representing task preconditions. An HTN-planning problem (d, s, D) is *totally ordered* if d and all the task networks in the methods of D are totally ordered. This corresponds to the form of task network that the SHOP system handles.

We show an encoding of totally ordered HTN-planning problems in Golog. Consider an HTN-planning problem $P = (d, S_0, D)$. For each compound task h, with a set of methods $(h, d_1), (h, d_2), \ldots, (h, d_k)$ in D, we define the following Golog procedure:

proc h
$(\pi \vec{x}_1)[(L_{1,1})?\ ;\ t_{1,1}\ ; \ldots ; (L_{1,i_1})?\ ;\ t_{1,i_1}]\ |$
$(\pi \vec{x}_2)[(L_{2,1})?\ ;\ t_{2,1}\ ; \ldots ; (L_{2,i_2})?\ ;\ t_{2,i_2}]\ |$
$\ldots$
$(\pi \vec{x}_k)[(L_{k,1})?\ ;\ t_{k,1}\ ; \ldots ; (L_{k,i_k})?\ ;\ t_{k,i_k}]$
endProc

where $t_{i,j}$ is the jth task in d_i according to the ordering, $L_{i,j}$ is the conjunction of the literals l such that $(l, t_{i,j})$ is a constraint in d_i and $\vec{x}_i$ are the variables that appear free in the constrains and tasks of method d_i. Intuitively, the procedure non-deterministically chooses one of the methods to execute. Each line in the procedure non-deterministically chooses values for the free variables and then executes the sequence of precondition tests and tasks.

Let Δ_P denote the resulting set of Golog procedure declarations. To complete the encoding of the HTN-planning problem P we include a Golog program δ_d obtained from the task network d. This program has the same form as the subprogram for a single method:

$$(\pi \vec{x}).(L_1)? \; ; \; t_1 \; ; \ldots ; \; (L_n)? \; ; \; t_n.$$

EXAMPLE 4. The procedure in Example 2 is in fact a simplified version of the encoding of the methods in Example 3. The procedure that results by following exactly the encoding described above is as follows.

proc $moveObj(o, loc)$
  $(\pi \; oloc, ocity)[$
    $(atObj(o, oloc) \wedge inCity(oloc, ocity) \wedge inCity(loc, ocity))? \; ;$
    $inCityDeliver(o, oloc, loc)] \; |$
  $(\pi \; oloc, ocity, dcity, oap, dap)[$
    $(atObj(o, oloc) \wedge inCity(oloc, ocity) \wedge inCity(loc, ocity) \wedge$
      $inCity(loc, dcity) \wedge inCity(oap, ocity) \wedge inCity(dap, dcity))? \; ;$
    $inCityDeliver(o, oloc, oaprt) \; ;$
    $airDeliver(o, oaprt, daprt) \; ;$
    $inCityDeliver(o, daprt, loc)]$
endProc

Given the above encoding of (totally ordered) HTNs in Golog, we can formulate a specification of the HTN-planning problem in terms of the axiomatization of primitive tasks (a basic action theory) and Golog, and of logical entailment, as follows. Let P be a totally ordered HTN-planning problem, then

$$\mathcal{D}_P \models (\exists s) Do(\Delta_P \; ; \; \delta_d, S_0, s). \tag{1}$$

Here, $\mathcal{D}_P$ is the basic action theory of P and it would also include the axioms defining the semantics of the Golog constructs shown in Section 2.2. However, we would like to remark again that the logical formalization of Golog is much simpler than that of ConGolog and the axiomatization of $Trans$ and $Final$ is not really necessary. This means that, unlike the case of partially ordered HTN-planning, the formalization of totally ordered HTN-planning requires only a standard basic action theory in the Situation Calculus plus a set of macros for the Golog constructs.

The following theorem formalizes the relationship between the operational semantics of totally ordered HTN-planning and the Golog encoding described above. For a sequence σ of ground primitive tasks, i.e. ground action terms $\alpha_1, \ldots, \alpha_n$, let $do(\sigma, s)$ denote the situation term $do(\alpha_n, do(\alpha_{n-1}, \ldots, do(\alpha_1, s) \ldots)$.

THEOREM 5. *Let $P = (d, S_0, D)$ be a totally ordered HTN-planning problem, Δ_P and δ_d be the corresponding Golog procedures and program. Then*

$$\sigma \in sol(d, S_0, D) \textit{ iff } \mathcal{D}_P \models Do(\Delta_P\ ;\ \delta_d,\ S_0, do(\sigma, s)).$$

3.2 Partially ordered task networks

Let us next considered partially ordered HTNs. The only difference with the HTNs of the previous subsection is that the boolean formulae ϕ in task networks specify through constraints $t \prec t'$ a partial instead of a total ordering on the tasks. Thus, formulae ϕ will be assumed to be arbitrary conjunctions of constraints of the form $t \prec t'$ and (l, t).

This form of HTN corresponds to the HTNs the SHOP2 system handles, except that SHOP2 additionally allows so-called *protection requests* and *protection cancellations*. These are used, for example, to ensure that some condition be maintained from the time one task finishes executing and another starts. These constraints are not difficult to add to our encoding, however, they are a side issue which for clarity of presentation we prefer not to address here.

The main motivation behind allowing the tasks to be partially ordered is the possibility of allowing multiple tasks to execute concurrently, that is, to allow their subtasks to interleave when there is no precedence ordering specified between them. Interleaved concurrency is exactly the type of concurrency that the construct $\|$ of ConGolog provides.

Let us then consider encoding partially ordered HTN-planning problems in ConGolog. As before, for each compound task we will define a procedure. But in this case we will need to introduce some auxiliary fluents and actions that will help enforce the task precedence constraints.

First, we need a fluent $terminated_{tname}(\vec{x}, s)$ for every compound task $tname(\vec{x})$ or primitive task (action) $A(\vec{x})$. Given a task, let τ stand for symbol $tname$ if a compound task and for A if a primitive task. Intuitively, $terminated_\tau(\vec{x}, s)$ holds when task $\tau(\vec{x})$ has already executed and terminated in situation s. We also need to introduce an auxiliary primitive task $end_\tau(\vec{x})$. For compound tasks, $end_\tau(\vec{x})$ will be added as a sub-task of $\tau(\vec{x})$ and will be the last sub-task to execute. The intention obviously is that this task will set the fluent $terminated_\tau(\vec{x}, s)$ to true when $\tau(\vec{x})$ finishes executing.

Formally, the successor state axioms for fluents $terminated_\tau(\vec{x}, s)$ are as follows:

$$terminated_\tau(\vec{x}, do(a, s)) \equiv \quad a = \tau(\vec{x}) \vee a = end_\tau(\vec{x}) \vee terminated_\tau(\vec{x}, s).$$

These fluents are all initially false, i.e. $(\forall x)\neg terminated_\tau(\vec{x}, S_0)$ is included in the initial database $\mathcal{D}_{S_0}$. Since these fluents and the actions $end_\tau(\vec{x})$ are auxiliary "system actions," so to speak, they can be kept transparent from the "user".

Before we introduce the ConGolog encoding, we need to introduce one last notation. Let (T, ϕ) be a task network and $\tau(\vec{x})$ be one of its tasks. We define $pred_\tau(\vec{x}, s)$ as the following conjunction:

$$pred_\tau(\vec{x}, s) \stackrel{\text{def}}{=} \bigwedge_{\{\tau' : (\tau'(\vec{x}') \prec \tau(\vec{x})) \in \phi\}} terminated_{\tau'}(\vec{x}', s).$$

If there is no constraint $(\tau'(\vec{x}') \prec \tau(\vec{x}))$ in ϕ then $pred_\tau(\vec{x}, s) \stackrel{\text{def}}{=} True$. Intuitively, $pred_\tau(\vec{x}, s)$ holds in s if all tasks which precede τ according to the constraints have already executed.

We are now ready to present the encoding. For the sake of readability we omit term arguments. The ConGolog procedure that encodes the methods $(h, d_1), (h, d_2), \ldots, (h, d_k)$ for a compound task h is:

proc h
  $(\delta_1|\delta_2|\ldots|\delta_k)$;
  end_h
endProc

where each δ_i stands for a program as follows:

$$\delta_i \left\{ \begin{array}{ll} (\pi\vec{v})\{ & [(pred_{t_{i,1}})?\ ;\ (L_{i,1})?\ ;\ t_{i,1}]\ \ \| \\ & [(pred_{t_{i,2}})?\ ;\ (L_{i,2})?\ ;\ t_{i,2}]\ \ \| \\ & \ldots \\ & [(pred_{t_{i,k_i}})?\ ;\ (L_{i,k_i})?\ ;\ t_{i,k_i}] \quad \} \end{array} \right.$$

The $t_{i,j}$ are the tasks in d_i and $L_{i,j}$ is as before the conjunction of the literals l such that $(l, t_{i,j})$ is a constraint in d_i. Intuitively, each δ_i encodes the task network d_i of the corresponding h method. The program δ_i essentially executes all the subtasks concurrently. But each subtask must first successfully test that all the subtasks that precedes it have already terminated, before it can execute. The semantics of construct $\|$ captures the intended meaning that the execution of a task is suspended until the preceding tasks, according to the ordering constraints, have executed.

One aspect of the above encoding is not completely faithful to the HTN semantics of [Erol, Hendler, and Nau 1996]. The intended meaning of a constraint (l, t) is that the literal l must hold immediately before task t executes. In the above ConGolog encoding, the corresponding code is $l?\ ;\ t$. This encoding and interleaved concurrency with other tasks means that some other tasks may execute after the test $l?$ and before the execution of t. It would not be difficult to add to ConGolog a new programming construct for synchronized *test-and-execute* or use interrupts as in [Gabaldon 2002]. However, requiring a condition to hold immediately before a task begins executing seems to us to make sense only for primitive tasks, since the subtasks of a compound task would be interleaved anyway. Then in the case of a primitive task, a condition required to hold before the task executes can be encoded in the action precondition axiom instead, since primitive tasks are actions from the basic action theory. We will therefore proceed with the relaxed form of the constraint.

EXAMPLE 6. This is a simple blocks world example method for moving a block v_1 from on top a block v_2 onto a block v_3:

$$\begin{array}{l} (move(v_1, v_2, v_3) \\ \ clear(v_1),\ clear(v_3),\ unstack(v_1, v_2), stack(v_1, v_3) \\ \ (clear(v_1) \prec unstack(v_1, v_2)) \wedge (clear(v_3) \prec unstack(v_1, v_2)) \wedge \\ \ (unstack(v_1, v_2) \prec stack(v_1, v_3)) \\ ) \end{array}$$

The encoding as a ConGolog procedure is as follows. Assuming the above method is the only method for task $move(v_1, v_2, v_3)$, the procedure declaration is:

proc $move(v_1, v_2, v_3)$
  δ_1;
  $end_{move}(v_1, v_2, v_3)$
endProc

where δ_1 is the following program (the tests $(True)?$ can obviously be removed):

$(True)?\ ;\ clear(v_1)\ \|$
$(True)?\ ;\ clear(v_3)\ \|$
$(terminated_{clear}(v_1)\ \wedge terminated_{clear}(v_3))?\ ;\ unstack(v_1, v_2)\ \|$
$(terminated_{unstack}(v_1, v_2))?\ ;\ stack(v_1, v_3)$

If the options of writing task networks or ConGolog programs are both available, in some cases it may be easier to write ConGolog directly instead of an HTN and then translate. Writing ConGolog directly had an additional advantage that the direct encoding may require a simpler theory compared to the result of an automatic translation of an HTN into ConGolog. The reason is that there may be less overhead in the form of auxiliary fluents and actions. For instance, consider again the method for $move(v_1, v_2, v_3)$. This could have been encoded as the following much simpler ConGolog program:

$(clear(v_1)\ \|\ clear(v_3))\ ;$
$unstack(v_1, v_2)\ ;$
$stack(v_1, v_3)$

On the other hand, for more complex methods it may be much easier and more natural to write instead a task network with partial ordering constraints. In that case, ConGolog procedures can be obtained automatically by applying the above encoding.

The logical specification (1) of HTN-planning is the same in the case of partially ordered HTNs encoded in ConGolog as shown above. The only difference is that in this case, the theory $\mathcal{D}_P$ must include the axiomatization of the $Trans$ and $Final$ relations, so it is necessarily a more complex theory.

A similar result on the correspondence between the operational semantics of HTNs and the ConGolog encoding can be obtained.

THEOREM 7. *Let $P = (d, S_0, D)$ be a partially ordered HTN-planning problem, Δ_P and δ_d be the corresponding ConGolog procedures and program. Then*

$$\sigma \in sol(d, S_0, D) \textit{ iff } \mathcal{D}_P \models Do(\Delta_P\ ;\ \delta_d,\ S_0, do(\sigma, s)).$$

4 On-line Execution with Exogenous Actions

As we mentioned earlier, one of the benefits of the encoding is that it brings to HTNs a number of additional features that already exist for the Golog family of languages. The purpose of this section is 1) to show some sample runs obtained with the Prolog interpreters of ConGolog and some translated HTNs shown earlier, and 2) to demonstrate one particular

feature of the Golog family of languages that is normally not available in HTN systems, namely, online execution with exogenous events. To this end, we will resort again to the logistics domain example and its ConGolog encoding, and extend it for modeling online execution with exogenous events in the form of run-time package delivery requests. Of course, once a set of HTNs has been translated into ConGolog, features like exogenous actions become an orthogonal issue. Nevertheless, this section also illustrates the use of the translated logistics domain HTNs discussed earlier and shows a sample run obtained with the implementation.

Online execution of a ConGolog program means that once the next primitive action to execute 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 entails that once such an action is chosen, the ConGolog interpreter cannot backtrack since it is not possible to backtrack from actually executing an action in the physical world. This behaviour is in fact very easy to model with the interpreters by means of the Prolog cut operator. The offline interpreter from [De Giacomo, Reiter, and Soutchanski 1998] 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, De Giacomo et al. simply add a cut after a one-step transition of the program. This one-step transition involves, among other things, choosing the next primitive action to execute, according to the program and the current situation. Hence, an online interpreter should not backtrack after such a step. The modified rule of the online interpreter is as follows:

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

This rule results in an interpreter called *brave* because after choosing an action it immediately commits and executes it. Alternatively, a *cautious* online interpreter may check, *offline*, before committing to execute an action, whether it is possible for the remainder of the program to terminate successfully. This behavior is captured by the following rule:

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

Online vs offline execution interpreters are further discussed in [De Giacomo, Reiter, and Soutchanski 1998; Reiter 2001]. An extension of ConGolog that includes a programming construct for offline deliberation was introduced in [De Giacomo and Levesque 1999].

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, the standard assumption is that the agent has complete knowledge about the possible exogenous actions that

can occur and what their effects are. In other words, the background basic action theory includes precondition and successor state axioms for exogenous actions too, but the agent does not control those actions. In our logistics example, we will consider one exogenous action: $requestDelivery(obj, loc)$, intuitively meaning that a request to deliver obj to loc has been issued. In our Prolog implementation, we simulate the occurrence of these exogenous requests by having the interpreter prompt the user to input them. Instead of this interactive approach to exogenous actions, another possibility would be to have the exogenous actions generated at random.

Following [De Giacomo, Lesperance, and Levesque 2000], a special procedure, $exoProg$ models the "observation" of exogenous actions. This procedure executes concurrently with the main logistics procedure shown below. The $exoProg$ procedure is defined as follows using the ConGolog construct for interrupts:

proc $exoProg$
$(\pi e)(exoActionOccurred(e) \rightarrow e)$
endProc

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, a dummy action with no effects meaning that no exogenous action occurred, or $endSim$ which has no effect either but instructs the interpreter to stop requesting the user to enter exogenous actions.

The main logistics procedure, called $deliveryDaemon$, is a recursive program that reacts to the occurrence of exogenous actions by triggering the execution of a $moveObj(obj, loc)$ task. An exogenous action $requestDelivery(obj, loc)$ causes fluent $deliveryReq(pck, loc)$ to become true, which in turn causes an interrupt to fire.

proc $deliveryDaemon$
$(\exists pck, loc) deliveryReq(pck, loc) \rightarrow$
$(\pi pck, loc)\{deliveryReq(pck, loc)?\ ;$
$startDelivery(pck, loc)\ ;$
$moveObj(pck, loc)\ ;$
$endDelivery(pck, loc)\}$
$\parallel\ deliveryDaemon$
endProc

Note that the number of delivery requests that will need to be served concurrently cannot be anticipated. What the above procedure does when a delivery request arrives (the interrupt fires) is to concurrently start serving the request and recursively invoke itself. Through the recursive call the procedure continues to wait for the arrival of new requests. This is admittedly a bit of a hack. What we really would like to use is a fork construct, which is not (yet) available in the Golog family.

Finally, the main ConGolog program for running the simulation of the logistics domain with run-time delivery requests consists of the parallel execution of the logistics procedure and the exogenous actions procedure:

$$exoProg \parallel deliveryDaemon.$$

The full Prolog implementation is available online.[3] Here is a sample run:

```
[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 above are primitive tasks listed 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.

[3] http://centria.di.fct.unl.pt/~ag/exologistics/

5 Conclusion

Our main goal in this chapter was to look at the relationship of HTN-planning and the Golog family of high-level action programming languages. We have shown that it is possible to encode HTN-planning problems into these languages without adding new constructs. In particular, totally ordered HTNs in Golog and partially ordered HTNs, which allow tasks to execute concurrently, in ConGolog. We showed that the operational semantics of an HTN-planning problem, i.e. the set of solutions, corresponds to the set of execution traces derived from the Situation Calculus formalization and (Con)Golog. We also showed a logical specification of the HTN-planning problem in terms of logical consequence from a basic action theory and the (Con)Golog axioms. Furthermore, we illustrated how through the translation one can combine HTNs with other features of the Golog family by taking the logistics domain example, adding exogenous delivery requests and running it on an online interpreter.

As we discussed in the introduction, HTNs and high-level action languages like Golog are two complementary ways of incorporating domain specific knowledge into automated planning. We also mentioned a third alternative that has been introduced by Bacchus and Kabanza [2000]. In their approach, domain specific knowledge is expressed in terms of Linear Temporal Logic, and is used to instruct their planning system, TLPlan, what alternatives *not* to explore. There is a consensus that this type of control knowledge and procedural knowledge as in HTNs and ConGolog, are both useful. Although we did not consider TLPlan style control knowledge here, we have considered it elsewhere [Gabaldon 2003; Gabaldon 2004]. In that work, we introduced a procedure for "compiling" TLPlan style control knowledge into a Situation Calculus basic action theory in a way that achieves the same pruning of unpromising plans as in TLPlan. That work together with our encoding of HTNs presented here, amount to a combination of both types of control knowledge into the single formal framework of the Situation Calculus. This combination is much further explored and developed in [Sohrabi, Baier, and McIlraith 2009].

References

Bacchus, F. and F. Kabanza [2000]. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence 116*, 123–191.

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

De Giacomo, G. and H. J. Levesque [1999]. An incremental interpreter for high-level programs with sensing. In *Logical Foundations for Cognitive Agents: Contributions in Honor of Ray Reiter*, pp. 86–102. Springer.

De Giacomo, G., R. Reiter, and M. Soutchanski [1998]. Execution monitoring of high-level robot programs. In A. Cohn and L. Schubert (Eds.), *6th International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pp. 453–465.

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

Gabaldon, A. [2002]. Programming hierarchical task networks in the situation calculus. In *AIPS'02 Workshop on On-line Planning and Scheduling*, Toulouse, France.

Gabaldon, A. [2003]. Compiling control knowledge into preconditions for planning in the situation calculus. In G. Gottlob and T. Walsh (Eds.), *18th International Joint Conference on Artificial Intelligence (IJCAI'03)*, pp. 1061–1066.

Gabaldon, A. [2004]. Precondition control and the progression algorithm. In D. Dubois, C. Welty, and M.-A. Williams (Eds.), *9th International Conference on Principles of Knowledge Representation and Reasoning (KR'04)*, pp. 634–643.

Levesque, H., F. Pirri, and R. Reiter [1998]. Foundations for the situation calculus. *Electronic Transactions on Artificial Intelligence 2*(3-4), 159–178. http://www.ep.liu.se/ej/etai/1998/005/.

Levesque, H. and R. Reiter [1998]. High-level robotic control: beyond planning. Position paper. In *Cognitive Robotics AAAI Fall Symposium*, pp. 106–108.

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

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., H. Munoz-Avila, Y. Cao, A. Lotem, and S. Mitchell [2001]. Total-order planning with partially ordered subtasks. In B. Nebel (Ed.), *17th International Joint Conference on Artificial Intelligence (IJCAI'01)*, pp. 425–430.

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

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

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.

Sohrabi, S., J. A. Baier, and S. A. McIlraith [2009]. HTN planning with preferences. In C. Boutilier (Ed.), *21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, pp. 1790–1797.

Tate, A. [1977]. Generating project networks. In R. Reddy (Ed.), *5th International Joint Conference on Artificial Intelligence (IJCAI'77)*, pp. 888–893.