Generalizing Problem Reduction: A Logical Analysis *

Drew McDermott
Yale University
Department of Computer Science

Abstract

Froblcm reduction is the name given to the problem-solving paradigm in which the problem solver manages a network of "tasks" representing its intentions, repeatedly reducing tasks to subtasks and coordinating their execution. This idea needs a lot of generalization for it to be able to handle a realistic range of problems. Even after the model of time is made more realistic (to handle continuity and branching), issues remain regarding what it means to have a task or a subtask, how a task can succeed or fail, whether a task is feasible. A profitable way to study these issues is to attempt to add axioms about tasks to a first-order temporal logic. The result sheds light on what sorts of generalizations of task networks are needed.

1. Introduction

Problem solvers like NOAH [Sacerdoti 75], NASL [McDermott 78), and SIPE [Wilkins 82] do what is called *problem reduction.* They work on problems, or *tasks,* by retrieving *plans* from some kind of *plan library.* Each task gives rise to a plan, which consists of one or more *subtasks.* The subtasks are actions, which, if done in the appropriate order, will solve the original problem. Each subtask is either *primitive,* or becomes a new problem. The structure of partially ordered tasks is called a procedural network, or *task network.* See Figure 1-1

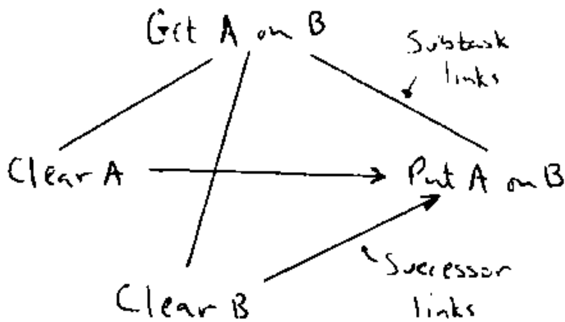

Figure 1-1: Task Network for Blocks-World Problem

This work was supported by the National Science Foundation under contract MCS-8203080

The subtasks interact in various ways, and may require reordering. For instance, if one task makes false a fact that another task requires to be true, it may be important to make sure that the falsifying task is ordered so that it occurs when the interaction no longer matters.

This framework is substantially that pioneered by Sacerdoti and explored by others since. Surprisingly little progress has been made in pushing it much further. One reason for this is that most problem solvers have had a consistently impoverished vocabulary for expressing temporal concepts. For instance, it has usually been assumed that nothing happens unless the problem solver makes it happen, and that each action the problem solver performs can be expressed in terms of "addlists and delotelists," which specify a finite, often context-independent, set of atomic facts that change in truth value instantaneously when the action is executed. These restrictions have historically been associated with applications of McCarthy's situation calculus [McCarthy .58], even though nothing in that calculus really requires them. (The use of "dynamic logic," as in [Rosenschein 81], freezes these faulty assumptions in an elegant crystalline form, but makes it harder to go beyond them than McCarthy's original formulation.)

Another reason for lack of progress is that the diagram of Figure 1-1 is too seductive. It implies that any action can be reduced to a network of subactions and arrows. In fact, this vocabulary for describing plans is quite weak. Of course, we can augment the vocabulary as much as we want; the trick is to retain the transparency that allowed Sacerdoti's NOAH to reason about what it intended to do.

As an example of the sort of reasoning a problem solver should be able to do, suppose that it is set the task of managing a water supply. It can fill a main supply tank by opening an inlet, up to some level, and must be prepared to use water from the supply for various purposes. In particular, the water will be needed for a series of industrial tasks (cooling things, putting out fires, or whatever). We can model this as a sequence of two tasks: Open the valve, and, for each event that requires water, draw down the required amount from the tank.

Beyond this point, problem solvers have not been able to perform problem reduction. And yet, it seems a matter of simple temporal reasoning to foresee how much water will be available and roughly how much will be needed. If there is a discrepancy, it should be noticed (by a "critic"), and task-network revision should ensue.

In [McDcrmott 82], I tried to enlarge the vocabulary for talking about time and actions, to allow actions like, "Turn on the water," "Allow the tank to fill," "Avoid leaving the room," and "Prevent the tub from overflowing." None of these could be expressed in the older formalism. This was accomplished by enriching the ontology of the situation calculus. Situations, now thought of as instantaneous *states* of the universe, were assumed to be packed into continuous sequences. The notion of a "next" state was abandoned. Even while the problem solver does nothing, time continues to advance. This allows one to reason about processes and agents outside the problem solver, which I shall refer to as "the robot" (since it reasons about taking actual actions in the real world).

Another important augmentation was to stipulate that universe states are partially ordered. Time branches into the future, so two states can both be in the future of "now'', and not be comparable; they represent alternative futures. In general, the outcome of an action is not a single state as has often been assumed. Since lots of other things can be happening at the same time, and even simple actions have unpredictable effects, an action is thought of as happening over an interval. We write this (Occ s1 *s2* (do Robot *A)):* from state ,sl to state *s2,* the robot does action *A.* The same action can be done in many different intervals, all starting in s1. Stated otherwise, many different states of the world can result from doing a given action in a given state.

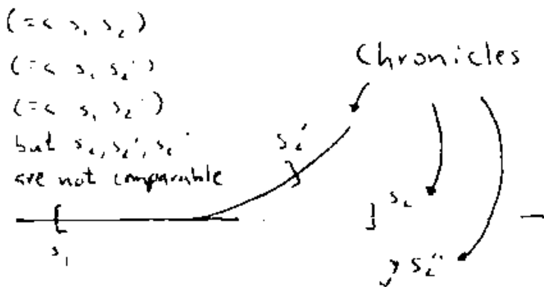

Figure 1-2: Indeterminacy of Events

This is a promising start, but only attacks half the problem. We can talk about actions in a more sophisticated way (and reason, for instance, about tanks filling up, as in [McDermott 82]). We must now develop ways of talking about problem-solver intentions as well.

It should be clear that in this paper I am exploring representational issues by the formal-logical methods pioneered by McCarthy [McCarthy 58] and Hayes [Hayes 79]. Hard-nosed problem-solving researchers may be impatient with this. To many of them it must seem a waste of time to work on anemic logical studies when red-blooded arms and motors await. Eventually, however, the *ad hoc* nature of the problem solvers we build will catch up with us. We may as well act now to develop temporal calculi that can support the kinds of reasoning we will need to build into our autonomous robots.

2. Tasks and Subtasks

We must add to the temporal logic the ability to talk about tasks and plans as well as simple actions. The basic notion is, of course, the fact of having the intention to do an action. We write this as (T *8* (task *k A)).* (T *s p)* means that fact *p* is true in state s; this is just a variant of the situation calculus, (task *k A)* is a particular fact, true in some states and false in others. It is true if the robot has the intention of doing *A* whenever it can. *k* is a term denoting this intention. We have axioms like this:

Axiom 1:

```
(if (T ?s (task ?k ?a))
    (= ?a (task-act ?k)))
```

"Every task has a unique, unchanging action, denoted (task-act *task*)."

As before, I write logical formulas in a LISP-like syntax. We have the usual logical connectives, "and," "or," "if," and "not." Variables universally quantified throughout an entire formula are indicated by prefixing all of their occurrences with "?" . Other quantifiers are indicated by (forall (-*vars*-) *formula*) and (exists (-*vars*-) *formula*)

Axiom 2:

```
(if (and (T ?s (task ?k1 ?a))
         (T ?s (task ?k2 ?a)))
    (= ?k1 ?k2))
```

"There is just one task for a given action at a given time."

We define (is-task *k*) to be true just when *k* is a task:

Definition 1:

```
(iff (T ?s (is-task ?k))
     (exists (a)
        (t ?s (task ?k a))))
```

iff ("if and only if") is an *n*-place connective asserting that two or more formulas are equivalent.

A task remains a task for a single uninterrupted interval:

Axiom 3:

```
(if (and (T ?s1 (is-task ?k))
         (T ?s2 (is-task ?k))
         (=< ?s1 ?s2))
    (forall (s)
       (if (< ?s1 s ?s2)
           (T s (is-task ?k)))))
```

If a state precedes another, we write this as (< *s*1 *s*2). The symbol =< is used to include the case where they are identical. Two states are said to be in the same *chronicle* if

they are comparable (identical or one preceding the other); this is written (>=< s1 s2). See Figure 1-2.

At a given time, the problem solver has a set of tasks that entirely define its intentions. For example, the problem solver might have the following tasks:

```
(T S0 (task T1
        (prog <(unscrew lightbulb1)
               (screwin lightbulb2)
               (discard lightbulb1)>)))

(T S0 (task T2 (screwin lightbulb2)))
```

That is, it is engaged in two things: a three-step plan, and the task of screwing in a lightbulb. This second task is step 2 of the three-step plan. This coincidence might be an accident: there might be some other reason for screwing in lightbulb2. If it is not an accident, we use the *subtask* fact-predicate to notate this:

```
(T S0 (subtask T2 T1 <2>))
```

This formula says that T2 is the second action of T1. The third argument to *subtask* is a "path expression," a tuple that unambiguously picks out a subpiece of an action. (Tuples are written using angle brackets; the empty tuple is denoted by <>.) This notion relies on an "abstract syntax" [McCarthy 62] for each action-description primitive. For example, *prog* takes a tuple of steps and denotes the action of doing one after another. Each subaction is indicated by a positive integer. Another example is (while *fact act*). In the action

```
(while (not (nail-driven))
       (repeat (prog <(lift hammer)
                      (drop hammer)>)))
```

we can indicate (lift hammer) using the path expression <act 1>, that is, the 1st step of the prog, which the *act* part of the while. The *test* part of the while is the action of testing whether or not the nail is driven. Hence, the path expression <test> is used for subtasks that do this test:

```
(T S0 (task K25 (look-at-nail)))
(T s0 (subtask K25 nail-drive-task <test>))
```

A subtask does not have to be derived from a supertask in this simple way. In fact, to transcend triviality, a working program must contain a mechanism (the "plan library") that supplies actions to carry out other actions when needed. If a subtask is derived this way, we make its path expression <>. So we might have

```
(T S0 (task T26 (replace lightbulb1
                         lightbulb2)))
(T S0 (subtask T27 T26 <>))
```

I will use the term *syntactic subtask* for a subtask with non-<> path expression; that is, for a subtask whose action is derived from the action of the supertask.

There are just three ways something can cease to be a task: success, failure, or "evaporation." The last category summarizes those cases when a task vanishes from the agenda because it is pointless, due to success or failure of all its supertasks. Consider the task "Poison Daddy Warbucks," which Orphan Annie might have as a subtask of "Get Daddy Warbucks's inheritance." The subtask succeeds if Warbucks is poisoned by Annie; fails if she is thwarted; and evaporates if Warbucks dies on his own.

We formalize this as follows:

Axiom 4:

```
(iff (Occ ?s ?s' (task-end ?k))
     (Occ ?s ?s'
          (become
             (not (is-task ?k))))
     (or (Occ ?s ?s' (succeed ?k))
         (Occ ?s ?s' (fail ?k))
         (Occ ?s ?s'
              (evaporate ?k))))
```

In this formula, the event (task-end *k*) is equated with *k*'s ceasing to be a task ("becoming not a task"), which can happen in one of the three ways mentioned. We can go on to provide axioms that specify what happens in each of these three cases.

Success is straightforward; a task succeeds the first time its action is done. Evaporation occurs only when the supertasks of a task go away:

Axiom 5:

```
(if (Occ ?s1 ?s2 (evaporate ?k))
    (forall (k')
       (if (exists (s p)
              (and (=< ?s1 ?s ?s2)
                   (subtask
                      k' ?k p)))
           (Occ ?s1 ?s2
                (task-end k')))
    ))
```

This axiom is not a biconditonal, because a problem solver can have tasks with no supertasks (e.g., "Stay alive").

One possible definition of failure is that a task has failed when its action becomes impossible. Rather than accept this definition, I opt for providing axioms explaining how every task of interest fails; more work is needed to see if the more general definition can be made to work.

3. Feasibility

In McCarthy's situation calculus, a typical deduction was of the form "Find a sequence of actions that transform situation *SO* into a situation in which fact *P* is true." Such deductions were done in an environment in which the only actions that were named were also feasible. That is, the axioms were set up so that only plans like "Put A on B, then put C on A" could be generated. A plan like "Play the horse that's going to win tomorrow" would never come up.

A significant flaw in this axiom is that it neglects the possibility that a task goes away because it ceases to be the best way to carry out its supertasks. Treating this case would require a substantial extension to the framework of this paper.

Modern problem solvers generate abstract intentions before concrete plans to carry them out. As their possibilities widen, it will get increasingly difficult to ensure that simply finding an action sequence's name will guarantee that it is feasible.

The task-subtask calculus gives us some hints how to analyze this important problem. The major idea is to analyze feasibility thus: An action is feasible if trying it would cause it to happen. We analyze trying something as having a task to do it.

The definition of feasibility involves a counterfactual or subjunctive conditional. (As in, "*A* was feasible, because if you had tried it, you would have succeeded.") Fortunately, the branching time we assumed makes this a rather easy counterfactual to handle. We simply suppose that a branch of the universe is taken in which a "test task" is injected into the robot's intention structure. If the "test task" would succeed, then the action is feasible.

One modification is needed here. Suppose the robot has the task, "Stay in this room." Then the action "Take this tool to the basement" is feasible, but only at the cost of upsetting the other task. So we introduce the notion of feasibility relative to a set of *boundary tasks,* none of which may be allowed to fail while the test task is performed.

We use the term (reltrytask *state action boundary-task-set)* to refer to the attempt to carry out *action* starting in *state* without upsetting any of the tasks in *boundary-task-set.* We suppose that the robot has "free will," and could essay a reltrytask on any action at any time. See Figure 3-1.

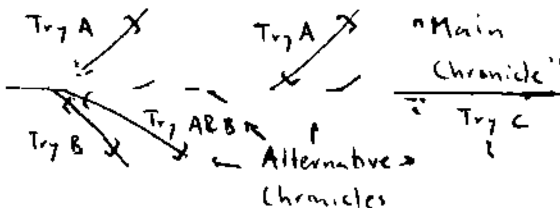

Figure 3-1: I Could Try to Do Anything, if I Wanted To

reltrytasks play a role like that of "test particles" in physics: hypothetical entities introduced into a situation that react to it without disturbing it. To determine if an action is feasible, we posit that it is tried, and see if we can deduce that it is successful.

We have to be quite careful about the way this is done, in order to avoid fallacies like this one: Suppose that *Al* and *A2* are both feasible, because if either is tried, it succeeds. Then suppose both are tried simultaneously. This event qualifies as a try of each separately, so we can conclude that both will happen, and hence that it is feasible that both can be done simultaneously. Since *Al* might be "leave the room," and *A2* might be "Stay in the room," you see the problem.

Therefore, we want to make the criterion be that an "isolated reltry" of an action would succeed. An *isolated* reltry is one that occurs without any other crazy tasks popping up, including especially other reltries. In the example of the previous paragraph, the proofs that *Al* and *A2* were feasible would depend on what happened when each was tried in isolation from the other. As desired, nothing could then be concluded about a situation in which both were tried at once.

We can't isolate a reltry too much, however, or the result will be useless; as soon as we put a complex task network around a task, the isolation condition will no longer hold, and feasibility will not allow us to conclude anything.

The following definition appears to do the job: An isolated reltrytask is one that takes place without any other new reltrytasks occurring, except subtasks and syntactic supertasks of the test task. For instance, in a proof that it is feasible to win the election of 1984, we will posit a reltrytask to win it. We forbid weird new reltrytasks like "Streak down Pennsyvania Avenue," but we allow subtasks like "File for candidate status by January, 1983."

With this definition of isolated task (see [McDermott 83) for the details), we can define feasibility as follows: *A* is feasible in state *S* with respect to boundary tasks *KK* if an isolated reltrytask beginning in *S* would succeed or evaporate without any element of *KK* failing.

We can use this definition to prove these theorems (see [McDermott 83]):

1. If *Al* is feasible in *SO,* and *A2* is feasible in every state resulting from doing *Al,* then (prog *<Al A2>)* is feasible in *SO.*
2. If an event is certain to happen before any boundary task in a given set fails, then waiting for it is feasible with respect to those boundary tasks.

It is interesting that to prove these theorems, it is necessary to be explicit about how the tasks involved might fail. For instance, one must state explicitly that:

> A task to do (prog *<A1 A2>)* is accomplished by a task to do *Al* followed by a task to do *A2,* and it can fail only if the current task fails.

Separating tasks out from the actions they call for has the advantage that one can talk about *Jailing* to accomplish something as well as accomplishing it.

Another important axiom that must be added to the system is that if an action is feasible, and is a way of carrying out another action, then the second action is feasible:

Axiom 8:

```
(if (and (T ?s (feasible ?a1 ?kk))
         (forall (s2)
            (if (Occ ?s s2
                     (do Robot ?a1))
                (Occ ?s s2
                     (do Robot
                        ?a2}))   ))
     (T ?s (feasible ?a2 ?kk)))
```

In this axiom, the fact of *A* being feasible with respect to boundary task set *KK* is written (feasible *A KK*).

4. An Example

In this section, I will sketch briefly an example showing the utility of these ideas. This is a chess problem I heard from John McCarthy.

I will call this state of affairs SO, although you must remember that this constant refers to an arbitrary snapshot of a board position that actually lasts until White makes a move; an uncountable set of other states go by during this time, during which the chess position doesn't change (although other things in the world will).

White can win, by the following argument: K can get to a5, because if k leaves the rectangle with corners c8 and g7 (an area 1 will call the "cage"), then the pawn at e6 will

```
                    a b c d e f g h
                  8 _|_|_|_|k|_|_|_ 8
                  7 _|_|_|_|_|_|_|_ 7
White to move     6 _|_|_|q|P|q|_|_ 6
                  5 _|_|q|P|_|P|q|_ 5
k = Black king    4 _|_|P|_|_|_|P|_ 4
K = White king    3 _|_|_|_|_|_|_|_ 3
q = Black pawn    2 _|_|_|_|K|_|_|_ 2
P = White pawn    1  | | | | | | |  1
                    a b c d e f g h
```

Figure 4-1: A Chess Problem

queen. But then K can get to b6, because if k is anywhere but c7, white can move to b6 in one step, and if k is at c7, white *can* move to a6, then b6 By similar arguments, K can get to c6, and then to either d6 or d7; and then the pawn at e6 can queen.

I will discuss an approach to part of McCarthy's problem within the framework I have outlined. The part I will be concerned with is step one, showing that the white king can get to a5. (The remaining steps are more straightforward.) The interesting thing about this step is that the reasoning is "continuous": it talks about the white king moving toward a5 while the black king moves around in the "cage," completely neglecting the fact that these moves occur as interleaved jumps.

The following plan can be shown to be feasible and allow the robot (playing White) to get his king to a5, or queen the pawn at e6:

```
(interrupt (move K a5)
           (outside k cage)
           (move P/e6 e8))
```

where the action (interrupt *al p a2)* is defined thus:

Definition 2: (interrupt *al p aS)* is executed whenever one of the following happens:

1. *al* is executed without *p* becoming true.
2. p becomes true before *al* has been executed, and *a2* is then executed

This is the sort of thing that the original task networks (see Figure 1-1) cannot express, but that human problem solvers execute as plans all the time.

Fortunately, we *can* analyze the "interrupt" plan as giving rise to subtasks. However, the subtasks are not always the same, or always foreseeable. If *p* never becomes true, then there will be just one subtask; if it does become true, there will be two.

The proof that (interrupt *Al P AS)* is feasible depends on the following Lemma:

Lemma 1: In a state *SO,* if *Al* is feasible with respect to boundary tasks *KK* so long as *P* remains false, and if *AS* is feasible in the first state in which *P* becomes true after *SO* (if any), then (interrupt *Al P AS)* is feasible with respect to *KK* in *SO.*

This statement can be proved using Definition 2, but that is not sufficient. That definition adequately defined what it means to actually execute (interrupt ...), but did not specify what it meant to have an intention to execute it. Since actions can be executed accidentally, the two are quite different. So we must provide an axiom like this:

Axiom 7: If, over an interval, a problem solver has a task *K* to perform (interrupt *Al P AS),* then either

1. there is just one subtask *Kl* to perform (until *P Al),* and *P* stays false; or
2. there are two subtasks, *Kl* as described, which succeeds when *P* becomes true, and *K2,* a task that begins as soon as *P* becomes true.

Furthermore, at any moment while *K* is a task, it fails only if the current subtask fails.

See Figure 4-2.

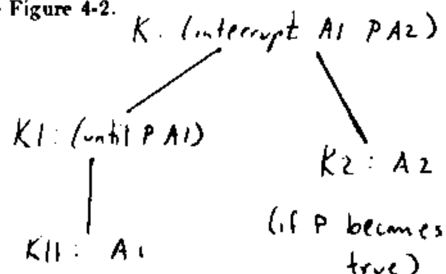

Figure 4-2: Structure of Interrupt Subtasks

In this axiom, I have had to introduce an intermediate task to perform the action (until *P Al).* The reason is that the subtask K11, with action *Al,* must evaporate if *P* becomes true, and therefore its su pert ask must end. Since the interrupt task itself can't end, we insert the until task, which succeeds if *P* becomes true:

Definition 3: (until *P A)* is executed over any interval in which *A* is executed without *P* becoming true (before the last instant), or over any interval in which P becomes true without *A* being executed.

The until-task has its own subtask structure (see Figure 4-2):

Axiom 8: Any task with action (until *P A)* has one subtask with action *A.* The su pert ask fails only if *A* fails before *P* becomes true.

Now we can prove the following theorem:

Theorem 2: (interrupt *Al P A2)* is feasible whenever *Al* is feasible provided *P* stays false, and *A2* is feasible in the first state after *P* becomes true.

Proof: Assume that an isolated reltry of (interrupt *Al P A2)* occurs. If *P* does not become true, then there is exactly one subtask *Kl* with action (until *P Al),* such that *K* fails only if *Kl* fails. But, by Axiom 8, there will be a unique subtask *K11* with action *Al.* Because *Al* is feasible, A*11* will succeed, and hence (Definition 3), *Kl* will succeed, and hence *K* will succeed. The proof for the case where *P* does become true is similar. QED

Several further steps are necessary to actually apply this theorem to the chess problem. Recall that White's plan is

```
(interrupt (move K a5)
           (outside k cage)
           (move P/e6 e8))
```

The basic strategy is of course to show that if k never leaves the "cage," then (move K a5) is feasible, and that if it does, then (move P/e6 e8) is feasible. While these are in some sense obvious, there are some pitfalls in the formal proof. For instance, how can we be sure that the blocked pawns never move, or that White (that is, the robot itself) doesn't move his pawn at e6 prematurely? These are "chess lemmas" which it is not necessary to prove (or not our job, anyway), but some care is necessary in stating them. The second issue especially raises interesting issues about predicting one's own subtasks. In Section , we had to restrict the "test task" for feasibility to be *isolated;* that is, no extraneous test tasks were allowed at the same time. We cannot rule out the robot's own genuine tasks so peremptorily; to prove feasibility, we must prove that no conflicting subtask will arrive. This is one reason proving feasibility is so difficult.

5. Conclusions

This paper has sketched an approach to reasoning about intentions within the framework of the temporal logic developed in [McDermott 82]. For a fuller treatment, see [McDermott 83]

Sections and showed the power of this calculus to illuminate interrelationships among tasks, feasibility, and possibility. In addition, they showed its flexibility in allowing us to talk easily of actions beyond the reach of previous problem solvers.

The same flexibility may carry over to task networks, allowing them to be generalized without losing their effectiveness. The original formalism assumed that a given task could be reduced to a foreseeable set of subtasks, linked by successor relationships. See Figure 1-1. We now have a more general picture. There are two sorts of subtask relationship: syntactic and non-syntactic. For instance, "Get A on B" might be reduced to the action

```
(prog <(parallel <(clear A) (clear B)>)
      (puton A B)>)
```

with three obvious subtasks, as shown in Figure 5-1.

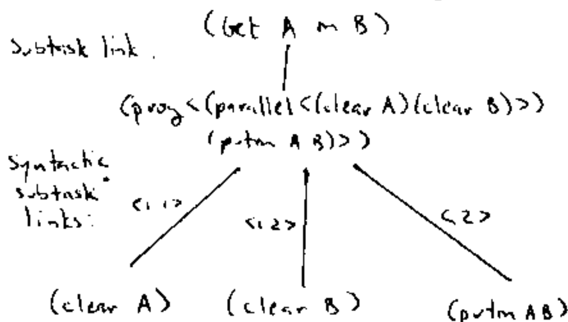

Figure 5-1: Revised Task Network Format

This picture is not very different from the previous one. One difference is that the new picture has no successor links, replacing them with labels on the syntactic subtask relationships. The successor-link notation always tantalized us with its non-generalizable transparency. The new notation is much more generalizable; any action, like prog, that can be defined in terms of subactions, can be used in such a net. For instance, an "interrupt" might have subtasks labeled <main> and <oops>. We call an action that is reduced syntactically a *macro-action.* A macro action with labeled subtasks replaces the successor link.

A second difference is that even the syntactic subtasks are not all foreseeable. This uncertainty is especially characteristic of macro-actions involving loops, (repeat *A* until *T)* may have zero or more subtasks with action *A,* with path expressions <1>, <2>, ... The number of subtasks is indefinite, but a problem solver can estimate how many there are going to be, and apply NOAH-style methods to their analysis. The new wrinkle is that the estimates can turn out to be wrong, an inconceivable possibility for NOAH. It is as yet unknown how to revise them; the method of [Doyle 79] may be useful.

Finally, a problem solver will want to keep track of different estimates, corresponding to different sets of interesting chronicles. For example, a task to do (interrupt At P A2) may have one or two subtasks, depending on whether P becomes true or not. If the system doesn't know whether P will happen or not, it may want to construct two different task networks, one for each eventuality. This operation may be desirable for almost any macro-action.

Acknowledgments: The ideas in this paper were developed in conversations with Robert Moore, Stan Rosenschein, Frnie Davis, John McCarthy, Stan Letovsky, and several others.

References

[Doyle 79] Doyle, J.
A truth maintenance system.
Artificial Intelligence 12:231-272, 1979.

[Hayes 79] Hayes, Patrick.
Ontology for Liquids.
1979.

[McCarthy 58] McCarthy, John.
Programs with common sense.
In Proceedings of the Symposium on the Mechanization of Thought Processes. National Physiology Laboratory, 1958.
In [Minsky 68], pp. 403-418.

[McCarthy 62] McCarthy, John.
Towards a Mathematical Theory of Computation.
In Proc. IFIP Congress 1962, pages 21-28. IFIP, 1962.

[McDermott 78] McDermott, Drew V.
Planning and acting.
Cognitive Science 2(2):71-109, 1978.

[McDermott 82] McDermott, Drew V.
A temporal logic for reasoning about processes and plans.
Cognitive Science 6:101-155, 1982.

[McDermott 83] McDermott, Drew V.
Reasoning about Plans.
1983.
To appear in Hobbs (ed.) Formal Theories of the Common-Sense World.

[Minsky 68] Minsky, M.
Semantic Information Processing.
MIT Press, Cambridge, Mass, 1968.

[Rosenschein 81] Rosenschein, Stanley J.
Plan Synthesis: A Logical Perspective.
In Proc. IJCAI 1981, pages 331-337. IJCAI, 1981.

[Sacerdoti 75] Sacerdoti, E.D.
A structure for plans and behavior.
Technical Report 109, SRI Artificial Intelligence Center, 1975.

[Wilkins 82] Wilkins, David.
Domain Independent Planning: Representation and Plan Generation.
1982.
SRI, 1982. Submitted to Artificial Intelligence.