Task-Decomposition via Plan Parsing

Anthony Barrett and Daniel S. Weld
Department of Computer Science and Engineering*
University of Washington, Seattle, WA 98195
{barrett, weld}@cs.washington.edu

Abstract

Task-decomposition planners make use of schemata that define tasks in terms of partially ordered sets of tasks and primitive actions. Most existing task-decomposition planners synthesize plans via a top-down approach, called *task reduction*, which uses schemata to replace tasks with networks of tasks and actions until only actions remain.

In this paper we present a bottom-up *plan parsing* approach to task-decomposition. Instead of reducing tasks into actions, we use an incremental parsing algorithm to recognize which partial primitive plans match the schemata. In essence, our approach exploits the observation that schemata are a convenient means for reducing search. We compile the schemata into a declarative search control language (like that used in machine learning research), which rejects plan refinements that cannot be parsed.

We demonstrate that neither parsing nor reduction dominates the other on efficiency grounds and provide preliminary empirical results comparing the two. We note that our parsing approach allows convenient comparison (and combination) of different search control technologies, generates minimal plans, and handles expressive languages (*e.g.*, universal quantification and conditional effects) with ease.

Introduction

NOAH (Sacerdoti 1975) introduced two important innovations: the partial order step representation and the use of schemata to define abstract tasks in terms of primitive actions and other tasks. NOAH represented plans as partially ordered sets of tasks and actions, and reduced tasks by substituting them with networks of tasks and actions until only actions remained. When tasks and actions interfered with each other, critic functions performed arbitrary plan-transformation repairs.

Despite the ubiquity of task-decomposition in "industrial strength" planners such as SIPE (Wilkins 1988, Wilkins 1992), and O-PLAN (Currie & Tate 1991), most formal analyses of planning have ignored the idea of hierarchies of decomposition schemata. Instead researchers focussed on NOAH's idea of partially ordered plan steps and dropped the notion of tasks. For example, TWEAK (Chapman 1987) and SNLP (McAllester & Rosenblitt 1991) synthesize plans solely from actions. Subsequent research continued the focus on actions, but explored expressive languages with conditional effects and universal quantification (Pednault 1988, McDermott 1991, Pednault 1991, Penberthy & Weld 1992). Only recently have formalists investigated task-decomposition (Yang 1990, Erol, Nau, & Hendler 1993), but their formulations assume the STRIPS action representation. Hence an interesting question remains:

> *Is there a sound and complete task-decomposition algorithm for an expressive language such as* ADL *(Pednault 1989), i.e. one with conditional effects and universal quantification?*

Inadequate emphasis on task-decomposition has also had an unfortunate effect on machine learning research for planning, *e.g.* explanation-based learning (Minton 1988), static domain analysis (Etzioni 1993, Smith & Peot 1993), abstraction (Knoblock 1990, Yang & Tenenberg 1990), case-based planning (Hammond 1990), and derivational analogy (Veloso & Carbonell 1993). It is unfortunate that the vast majority[1] of research on speedup learning has ignored task-decomposition planners, since defining and using tasks provides a very successful form of search control. This leads to a second question:

> *Can existing speedup learning techniques be used with task-decomposition planners?*

In this paper we describe the implemented UCPOP+PARSE algorithm, a new approach to task-decomposition that answers these questions. Instead of performing task-decomposition by *task reduction* UCPOP+PARSE inverts the process into *plan parsing*. In our algorithm, a partial-order planner synthesizes networks of actions, and an incremental parser identifies the decomposition that corresponds to the actions. We demonstrate that neither the reduction nor parsing approach strictly dominates the other on efficiency

*We appreciate helpful comments and suggestions from Denise Draper, Oren Etzioni, Keith Golden, Nick Kushmerick, Ying Sun, and Mike Williamson. This research is funded in part by National Science Foundation Grant IRI-8957302 and Office of Naval Research Grant 90-J-1904

[1] The PRIAR system (Kambhampati & Hendler 1992) is a rare exception.

grounds. However, our parsing approach provides two advantages:

1. **Uniform search control framework:** Since our parsing critic is implemented as a set of declarative search control rules, they can be turned on or off and can be combined with other forms of control knowledge: domain-dependent hand coded rules or the output of machine learning algorithms. We present preliminary experiments demonstrating this claim.
2. **Lazy minimal expansion:** Our approach expands tasks into actions as those actions become needed to solve a problem. All and only those actions that are actually *useful* in achieving goals are added to the plan. This enables UCPOP+PARSE to easily handle actions with conditional effects

In the next sections we (1) introduce schemata for an example which illustrates universally quantified and conditional effects, (2) review task reduction, (3) describe plan parsing, and (4) compare the two approaches to task-decomposition.

Task-Decomposition Problems

We illustrate task-decomposition with an extension of the *briefcase domain* (Pednault 1988), which we encode with five primitive operators. `Carry` moves the briefcase and all of its contents, `open` and `close` act on the briefcase, while `put-in` and `take-out` respectively add and remove items from the briefcase if it is open (Table 1). Note the use of conditional and universally quantified effects in the `carry` action (see (Pednault 1989, Penberthy & Weld 1992) for the action semantics.).

`open()`	precondition :	$\neg open$
	effect :	$open$
`close()`	precondition :	$open$
	effect :	$\neg open$
`take-out`(x)	precondition :	$open$
	effect :	$\neg in(x)$
`put-in`(x)	precondition :	$\exists l\ at\text{-}b(l) \wedge at(x,l) \wedge open$
	effect :	$in(x)$
`carry`(l, to)	precondition :	$at\text{-}b(l) \wedge \neg open$
	effect :	$at\text{-}b(to) \wedge \neg at\text{-}b(l) \wedge \forall x\ in(x) \Rightarrow (at(x,to) \wedge \neg at(x,l))$

Table 1: Actions in the briefcase world

These operators, together with a description of an initial state and a desired goal, constitute a planning problem. For example, suppose that there are two items (besides the briefcase): a paycheck P and a dictionary D. All three are initially at home, and the paycheck is inside the briefcase. The goal is to have the paycheck at home and the dictionary at the office. Clearly, the following action sequence solves the problem:

`open(); take-out(`P`); put-in(`D`); close();`
`carry(`*home*`,`*office*`)`

Decomposition Schemata

A task-decomposition planner solves planning problems like the one above by exploiting a set of useful plan fragments called *decomposition schemata*. Intuitively, a schema specifies a coordinated set of tasks and actions that combine to solve common subproblems. (Yang 1990) provides a formal definition; here we illustrate them graphically. For example, Figure 1 shows the `move-to` decomposition schema.

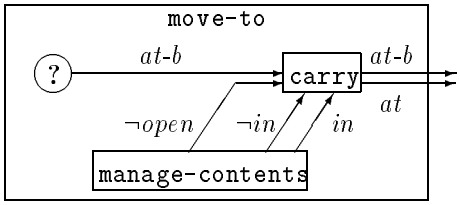

Figure 1: Decomposition schema defining a task for moving objects to a location.

The arrows leaving the box indicate that `move-to` is an appropriate way to achieve a goal involving either the *at* or *at-b* predicates. The contents of the schema indicate how these goals should be achieved: `move-to` expands into a `carry` action (defined above) and a `manage-contents` task (defined by another schema). Note that Figure 1 contains numerous arrows labeled with predicates (or their negation); these represent *protection intervals*. Each interval signifies that the producer (*i.e.* the node to the left of the arrow) is responsible for achieving literals involving that predicate for the *consumer* (the node on the right).[2] For example, the `manage-contents` task is responsible for achieving literals such as $\neg open$, $in(D)$, and $\neg in(P)$ for `carry`. Note that the information specified by a protection interval's producer is equivalent to a PRODIGY-style search control rule (Minton *et al.* 1989) which rejects attempts to achieve goals with inappropriate actions. As explained later, UCPOP+PARSE compiles schemata into a plan parser using rules of this form.

Figure 2 presents a schema which defines the `manage-contents` task. The intuition behind `manage-contents` is simple: first the briefcase is opened, then objects are added and removed from the case, finally it is shut once more. But the `manage-contents` schema illustrates an important feature that was absent from the previous example. This schema allows varying numbers of primitive `take-out` and `put-in` actions to be introduced. For example, if the `carry` action in the `move-to` schema requires that multiple objects be inside the briefcase, the `manage-contents` schema directs the planner to

[2] Intervals whose producer is drawn as a circle indicate that the corresponding literals can be achieved by reusing *any* existing action in the plan or by adding *any* new task that provides the literal. Readers familiar with NONLIN (Tate 1977) or O-PLAN (Currie & Tate 1991) should note that these "circle intervals" correspond to `achieve` conditions, while intervals whose producer is drawn as a box correspond to `supervised` conditions.

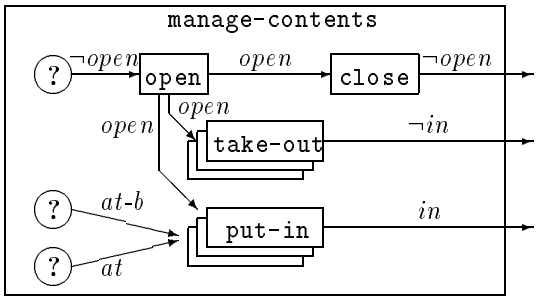

Figure 2: Decomposition schema defining a task for managing the briefcase's contents.

add as many `put-in` actions as necessary before the single `close` action.

Task Reduction

Typically task-decomposition planners iteratively (1) use schemata to expand tasks into networks of other actions and tasks, and (2) test the resulting partial plans with a *critic function*. This cycle starts with a single problem task, and continues until only primitive actions remain or until the critic detects an irreconcilable interaction in the plan (in which case backtracking is necessary).

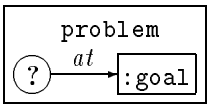

Figure 3: This top-level schema defines the language of appropriate plans. It specifies that *at* goals can be achieved in any way.

For example, Figure 3 shows how to expand a `problem` task that would correspond with the previously mentioned problem. The resultant plan can be reduced using the schemata shown previously. Figure 4 illustrates the final decomposition leading to the five step plan which solves the goal.

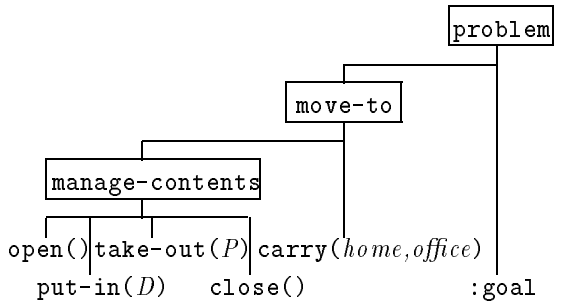

Figure 4: Decomposition to solve the briefcase problem

Since replacing tasks in accordance with the schemata yields a "legal" subset of the *possible* networks of actions, one can view a set of decomposition schemata as a *grammar* and see Figure 4 as a parse tree. The set of these schema-derived networks thus constitutes a formal language which we call the *schema-generated* plans. The set of all action sequences that solve a given planning problem (as defined by (Pednault 1991)) constitutes another language, which we call the *solution* set.

We adopt the insight (due to (Erol, Nau, & Hendler 1993)) that task-decomposition can be considered a search for a plan in the *intersection* of these two sets. Sound and complete partial-order planners such as TWEAK (Chapman 1987), SNLP (McAllester & Rosenblitt 1991), and UCPOP (Penberthy & Weld 1992) generate the solution space directly. Traditional task-decomposition planners use a *task reduction* process that expands the schema-generated set, then selects for membership in the solution set.

Plan Parsing in UCPOP+PARSE

Our approach breaks from this tradition by reversing the roles of the refinement and testing processes. Instead of reducing a task into a fixed network of actions and tasks, UCPOP+PARSE adds actions as their effects are needed and incrementally composes them into layers of tasks. Conceptually, UCPOP+PARSE can be thought of as the algorithm in Figure 5 which takes a problem specification as input and returns a plan.

Algorithm UCPOP+PARSE(*Problem*)

1. **Initialize:** Successively let:
   - P = `UCPOP-init-plan`(*Problem*) and
   - *Parses* = `init-parses`(P).
2. **Terminate:** If P solves *Problem*, return P.
3. **Refine:** Invoke `UCPOP-refine-plan`(P) and nondeterministicly choose P from the returned set.
4. **Parse:** Let *Parses* = `extend-parses`(*Parses*, P). If *Parses* becomes empty then fail.
5. Go to step 2.

Figure 5: The UCPOP+PARSE algorithm.

Actually, the schemata are compiled into a parser that interacts with the UCPOP planner (Penberthy & Weld 1992) through its general rule-based search controller. The routine `UCPOP-init-plan()` takes a problem specification and returns an initial plan consisting solely of a `:start` action (whose effects encode the initial conditions) and a `:goal` action (whose preconditions encode the goals). A plan that does not solve its problem contains at least one flaw; the UCPOP routine `UCPOP-refine-plan()` can take the plan, select a single flaw in the plan, and return a set of plans which have that single flaw repaired:

- If the flaw is an unsatisfied a precondition (*i.e.*, it is *open*), all effects that could possibly be constrained to unify with the desired proposition are considered. The returned set contains plans created by adding a single *causal link* (McAllester & Rosenblitt 1991) to the original plan. Each added causal link records a different way to satisfy the precondition using a new or existing action.

- If the flaw involves another action (called a *threat*) possibly interfering with the precondition being supported by the causal link, the returned set contains plans created by using methods to resolve the threat: either by ordering steps in the plan, posting additional subgoals, or by adding variable binding constraints.

For example, without guidance from the plan parser UCPOP solves the briefcase example (described previously) by generating the plan in Figure 6.

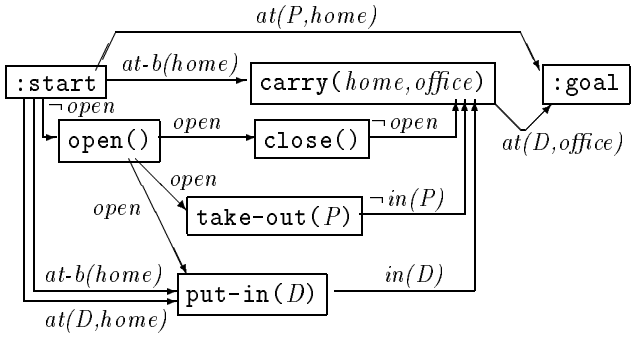

Figure 6: Primitive plan that solves the simple briefcase-world problem

The UCPOP planner is complete — if *any* solution to a planning problem exists, UCPOP will find it (Penberthy & Weld 1992). While this property has merits, it comes at the cost of forcing UCPOP to consider every conceivable way of achieving a goal, which in complex domains leads to poor performance. The decomposition schemata are compiled into the routines `init-parses()` and `extend-parses()`, which are used to prune inappropriate plans from UCPOP's consideration.

The routine `init-parses()` takes an initial plan and computes a set of parses. For example, in the briefcase problem, the routine would return a set with parse identifying the plan's `:goal` step with `:goal` node in the `problem` schema of Figure 3.

During the planning process in UCPOP+PARSE every plan under consideration has a set of parses associated with it. Each parse consists of a partially instantiated derivation tree and a mapping from actions in the plan to distinct leaves in the tree. The routine `extend-parses()` takes a plan, created by repairing a single flaw, and updates the set of parses to reflect that repair. When the repair adds a causal link, each parse must be extended to match the new link with a protection interval specified by the schemata in its derivation tree. Sometimes a match cannot be made, and a parse is deleted from the set. Other times more than one extension can be made to the parse. Thus, the number of parses varies as planning progresses.

As an example, consider the open goal *at(D,office)* in the briefcase problem. This flaw is repaired by adding a `carry` action. In this case, `:goal`'s associated `problem` schema provides little constraint, because the `at` link producer is a circle (signifying "anything goes"). The planning process becomes much more constrained by the parser when dealing with the open preconditions of the plan's `carry` action. The `carry` action must appear in a `move-to` task, and Figure 1 shows that the schema for `move-to` has strong constraints on its associated protection intervals. When UCPOP+PARSE returns the solution plan in Figure 6, it has a single associated parse (shown in Figure 4).

Comparing Parsing and Reduction

We compare the two approaches in terms of efficiency, ability to combine and evaluate multiple sources of search control knowledge, and the ability to handle action languages with universal quantification and conditional effects.

Efficiency

Since each approach performs a different type of least commitment, there are problems where one approach is preferable to the other. As illustration, consider the three schemata in Figure 7. In the top-level `problem` schema, the task `S` has to be decomposed using the other two schemata. `P` and `Q` are primitive actions, and while `P` and `S` do not explicitly affect α, `Q` asserts $\neg\alpha$. Since UCPOP+PARSE adds the step `Q` without committing to how `S` is decomposed to produce `Q`, it quickly determines that the problem is not solvable (`Q` necessarily clobbers α). A reduction planner, on the other hand, would loop forever since it can always reduce task `S`.

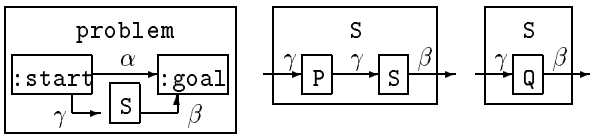

Figure 7: Schemata for a problem where plan parsing dominates.

In a slightly different problem the search space is finite for a reduction planner, but infinite for UCPOP+PARSE. Consider the three schemata in Figure 8. The only differences are the order of `S` and `P` in `S`'s decomposition schema, and the addition of $\neg\alpha$ to `S`'s explicit effects. Now the reduction planner determines that the problem is not solvable before reducing `S`, but UCPOP+PARSE can always add a new step `P` in the hope of a solution.

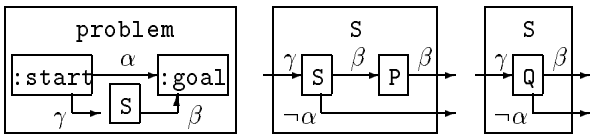

Figure 8: Schemata for a problem where task reduction dominates.

In general, the reduction approach can reason about the effects and preconditions of a task without committing to its decomposition, and the parsing approach can add a primitive step to a plan without committing to the manner in which tasks were decomposed

to produce that primitive step. Since there exist domains where each approach surpasses the other, the question becomes a matter of which planner performs best in practice. As a preliminary attempt to answer this question, we compared UCPOP+PARSE with a reduction planner that we implemented using the same routines to add new steps, handle protection intervals, and manage variable bindings. We performed several experiments in a NONLIN encoding of *blocksworld* and a house building domain. The results appear in Table 2.

problem	CPU seconds	
	UCPOP+PARSE	REDUCE
Sussman Anomaly	0.37	0.40
3 Block Tower Shift	2.10	1.33
Build House	0.83	0.23

Table 2: Performance of UCPOP+PARSE and a decomposition planner on two problem domains on a Sun SPARC IPX running Allegro Common Lisp.

In our experiments, both planners had similar performance, but that could be because both domains were simple; in the future we expect to perform further experiments with more complex domains.

Uniform Search Control Framework

One of the conceptual disadvantages to parsing is that it is less intuitive than reduction, with which people have twenty years experience. On the other hand, the ability to turn the parser on or off leads to a uniform framework for comparing other forms of search control with that provided by schemata.

To illustrate this feature, we tested UCPOP+PARSE on the example briefcase problem and by posing two problems in each of two more complex domains: *Tyreworld* (Russell 1992) and *Process Planning* (Gil 1991). In *Tyreworld*, the problems were to remove a tire from an automobile's hub, and to completely change a flat tire. The shortest solutions for these problems require 7 and 19 steps respectively. In *Process Planning* our problems involved sawing a block of copper and drilling a hole into a block of brass. These problems required 6 and 10 steps respectively.

The purpose of our experiment was to explore interactions between plan parsing and three different search strategies: vanilla best-first; best first search with a search space structured according to an abstraction hierarchy generated by the ALPINE machine learning algorithm (Knoblock 1990); and domain-dependent, hand-coded rules. To create our *Tyreworld* parser we defined 5 tasks for getting tools, inflating tires, removing tires, installing tires, and cleaning up. The *Process Planning* parser was created out of a set of 12 schemata defining tasks like setting up drill presses and securing objects to machines. The results of our experiments appear in Table 3.

Plan parsing clearly decreases the size of the search space, but the extent of the improvement depends on the difficulty of the problem relative to the search strategy being used. Note that in some cases, the amount of search reduction was insufficient to produce real speedup given the parsing overhead. Since the overhead depends on the number of possible parsings, and this is a function of the particular partial plan in question, parser overhead depends on the search strategy too. While we have only demonstrated that plan parsing is complementary with ALPINE, we believe that it can be usefully combined with explanation-based and other machine learning algorithms as well. We hope to validate these intuitions in future work.

problem: *strategy*	plans generated		CPU seconds	
	¬parse	parse	¬parse	parse
Briefcase:				
best first	151	41	1.1	0.4
Remove tire:				
best first	118	102	1.2	1.4
ALPINE	97	93	1.4	1.6
hand coded	134	118	1.8	1.9
Fix flat:				
best first	> 5000	702	> 103.6	17.5
ALPINE	> 5000	482	> 195.6	39.5
hand coded	579	308	14.7	8.6
Saw a block:				
best first	4820	941	135.3	29.1
ALPINE	4823	944	224.3	44.3
hand coded	434	292	9.9	6.9
Drill a hole:				
best first	> 5000	2854	> 142.3	94.4
ALPINE	> 5000	2855	> 272.5	136.8
hand coded	696	487	20.5	15.5

Table 3: Performance of UCPOP with and without parsing in three problem domains on a Sun SPARC IPX running Allegro Common Lisp. Problem runs marked with $>$ were terminated after a resource bound was exceeded.

Expressive Actions

We could not test our reduction planner in the *Briefcase*, *Tyreworld*, or *Process Planning*, domains because it can not handle actions with conditional effects. One of the assumptions typically made by reduction planners is that an action's preconditions are known when it is added to a plan during a reduction, and the reducing schema specifies how those preconditions are handled. This assumption does not hold when actions have conditional effects, since new preconditions may be introduced whenever an unused effect gets requested, or confronted. Extending a reduction planner to handle conditional effects is a topic for future research.

SIPE-2 (Wilkins 1992) does allow conditional effects and can compute universally quantified preconditions for an action, but the preconditions are computed when the action is added to the plan. It is unclear if it can be extended to add preconditions to a step at a later point.

Conclusions

We have described a fully implemented task-decomposition planner, UCPOP+PARSE, based on plan parsing. As it stands, UCPOP+PARSE can not handle

recursive schemata — they cause the parser to enter an infinite loop. We hope to soon adapt the flow-graph parsing algorithm of (Brotsky 1984) in order to circumvent this problem. We also wish to see if derivational analogy (Veloso & Carbonell 1993) can be used to automatically learn decomposition schemata.

We showed that neither the reduction nor parsing approach dominates the other in every problem domain. In addition, we reported preliminary experiments that suggest performance is comparable in the two approaches. However, plan parsing offers two advantages over the traditional task reduction method:

- Plan parsing incrementally expands a task into a variable number of primitive actions. This feature is useful for taking full advantage of actions with context dependent effects. All and only the actions that are actually needed in achieving goals are added to the plan.
- Because plan parsing acts to guide the search behavior of the UCPOP planner, we can combine and contrast the performance gains provided by decomposition schemata to that engendered by speedup learning and subgoaling mechanisms.

Since UCPOP has already been proven sound and complete (Penberthy & Weld 1992), UCPOP+PARSE automatically inherits soundness. In addition, it can be shown to be complete relative to the schema language intersection[3] as long as the parser is complete.

References

Brotsky, D. 1984. An algorithm for parsing flow graphs. AI-TR-704, MIT AI Lab.

Chapman, D. 1987. Planning for conjunctive goals. *Artificial Intelligence* 32(3):333–377.

Currie, K., and Tate, A. 1991. O-plan: the open planning architecture. *Artificial Intelligence* 52(1):49–86.

Erol, K., Nau, D., and Hendler, J. 1993. Toward a general framework for hierarchical task-network planning (extended abstract). In *Working Notes of the AAAI Spring Symposium: Foundations of Automatic Planning: The Classical Approach and Beyond.* Menlo Park, CA: AAAI Press.

Etzioni, O. 1993. Acquiring search-control knowledge via static analysis. *Artificial Intelligence* 62(2):255–302.

Gil, Y. 1991. A specification of process planning for PRODIGY. CMU-CS-91-179, Carnegie-Mellon University.

Hammond, K. 1990. Explaining and repairing plans that fail. *Artificial Intelligence* 45:173–228.

Kambhampati, S., and Hendler, J. 1992. A validation structure based theory of plan modification and reuse. *Artificial Intelligence* 55:193–258.

Knoblock, C. 1990. Learning abstraction hierarchies for problem solving. In *Proc. 8th Nat. Conf. on A.I.*, 923–928.

McAllester, D., and Rosenblitt, D. 1991. Systematic nonlinear planning. In *Proc. 9th Nat. Conf. on A.I.*, 634–639.

McDermott, D. 1991. Regression planning. *International Journal of Intelligent Systems* 6:357–416.

Minton, S., Carbonell, J. G., Knoblock, C. A., Kuokka, D. R., Etzioni, O., and Gil, Y. 1989. Explanation-based learning: A problem-solving perspective. *Artificial Intelligence* 40:63–118. Available as technical report CMU-CS-89-103.

Minton, S. 1988. Quantitative results concerning the utility of explanation-based learning. In *Proc. 7th Nat. Conf. on A.I.*, 564–569.

Pednault, E. 1988. Synthesizing plans that contain actions with context-dependent effects. *Computational Intelligence* 4(4):356–372.

Pednault, E. 1989. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. 1st Int. Conf. on Principles of Knowledge Representation and Reasoning*, 324–332.

Pednault, E. 1991. Generalizing nonlinear planning to handle complex goals and actions with context-dependent effects. In *Proc. 12th Int. Joint Conf. on A.I.*

Penberthy, J., and Weld, D. 1992. UCPOP: A sound, complete, partial order planner for ADL. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, 103–114. Available via FTP from `pub/ai/` at `cs.washington.edu`.

Russell, S. 1992. Efficient memory-bounded search algorithms. In *Proceedings of the Tenth European Conference on Artificial Intelligence.* Vienna: Wiley.

Sacerdoti, E. 1975. The nonlinear nature of plans. In *Proceedings of IJCAI-75*, 206–214.

Smith, D., and Peot, M. 1993. Postponing threats in partial-order planning. In *Proc. 11th Nat. Conf. on A.I.*, 500–506.

Tate, A. 1977. Generating project networks. In *Proc. 5th Int. Joint Conf. on A.I.*, 888–893.

Veloso, M., and Carbonell, J. 1993. Derivational Analogy in PRODIGY: Automating Case Acquisition, Storage, and Utilization. *Machine Learning* 10:249–278.

Wilkins, D. E. 1988. *Practical Planning.* San Mateo, CA: Morgan Kaufmann.

Wilkins, D. 1992. *Using the SIPE-2 Planning System, A Manual for SIPE-2 Version 4.* SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025.

Yang, Q., and Tenenberg, J. 1990. ABTWEAK: Abstracting a nonlinear, least-commitment planner. In *Proc. 8th Nat. Conf. on A.I.*, 204–209.

Yang, Q. 1990. Formalizing planning knowledge for hierarchical planning. *Computational Intelligence* 6(1):12–24.

[3] This corresponds to the definition of completeness used in (Erol, Nau, & Hendler 1993); an schema-based planner is likely to be incomplete with respect to the primitive solution set.