What Defaults can do that Hierarchies Can't

Matthew L. Ginsberg
CIRL
Department of Computer and Information Science
Department of Computer Science
University of Oregon
Eugene, OR 97403

Hugh W. Holbrook
Department of Computer Science
Stanford University
Stanford, CA 94305

Abstract. This paper extends existing work on the use of default information to describe planning hierarchies in two ways. First, we present a completeness result showing that all hierarchical planners can be described using defaults. Second, we show that if the planning hierarchy is situation-dependent, the default description is likely to have substantial computational advantages over a more conventional approach.

1 Introduction

There are a variety of arguments for planners that work declaratively instead of using a specialized description of action like that appearing in STRIPS [3] or SIPE [16]. One argument appearing in [9] is that declarative planners are best equipped to handle the ramification problem, since the description of actions with situation-dependent consequences is best handled inferentially. It is shown in [9] that attempting to deal with this problem by splitting actions into subtypes is equivalent to taking the deductive closure of the underlying declarative axioms and that the number of subtypes can be expected to grow exponentially with the size of the domain.

Nondeclarative systems have their advantages, however; one of the most important is that they support hierarchical planning. One of us has suggested in [6] that declarative planners can also work hierarchically by using the following technique:

1. Assume by default that every high-level plan can be expanded into a low-level one.

2. Using this assumption, find a plan that succeeds by default in achieving the goal in question.

3. Prove that the default assumptions used are monotonic consequences of the database. A constructive proof here will be equivalent to fleshing out the high-level plan into lower-level actions.

In this paper, we extend this idea in two ways. First, we show that every hierarchical system can be represented in this fashion, so that no expressive power is lost by viewing hierarchies as defaults. Second, we show that there is an analog to the ramification problem in this setting: If the planning hierarchy is situation-dependent, existing hierarchical planners will need to split actions into an exponential number of subtypes while a planner taking a declarative approach will not.

Before proceeding, however, we should be careful to stress that the claims we are making are representational only. Showing that declarative methods can overcome problems arising from the use of static hierarchies is something of a hollow result, since no declarative planning systems have been reported in the literature thus far. Progress is being made, however; there are now satisfactory nonmonotonic descriptions of actions and their consequences ([1] is typical) and one of us has proposed a specific declarative description that appears to have attractive computational properties [5]. Extending this work to a declarative planning system is touched upon in [7].

Our intended contribution here is more modest; we hope only to show that effective use of conventional planning hierarchies can require an exponentially large domain description, and that declarative methods can be used to overcome this problem. We feel that our arguments will add further weight to the view that practical planners will eventually need to be declarative systems, although it is certainly possible that existing nondeclarative methods can be modified to take advantage of the observations we will make.

There are two basic results in this paper, and they are the topics of the next two sections. In Section 2, we show that defaults can be used to describe existing planning hierarchies; in Section 3, we go on to show that the default approach is better able to handle hierarchies that are situation-dependent. Concluding remarks are contained in Section 4.

2 Hierarchical planning achieved with defaults

In this section, we introduce a simple planning problem that is best solved hierarchically. We present a typical hierarchical solution and then show that identical behavior can be obtained using defaults. We go on to show that the technique used is completely general, proving that a default description of hierarchy loses none of the computational benefits of hierarchical planning.

2.1 A sample domain

Consider a domain with a robot wandering through a maze of rooms that are connected by doors. Within a room, the robot may be located near any object or door in that room. The primitive actions that the robot can take include walking from room to room, going to a location within a room, and opening a door. The task of the planner is to generate plans for the robot to navigate through the maze to get to some particular object.

The domain actions appear in Table 1. The robot can walk through any door it is next to, provided that the door is open. It can open any door it is next to, and go

Action	Preconditions	Effects
`go-through`(d)	`in`$(r1)$ `door`$(d, r1, r2)$ `open`(d) `next-to`(d)	$\neg$`in`$(r1)$ `in`$(r2)$
`make-open`(d)	`in`$(r1)$ `next-to`(d) `door`$(d, r1, r2)$	`open`(d)
`go-to`(obj)	`location`(obj, r) `in`(r)	`next-to`(obj)

Table 1: Primitive actions of the robot

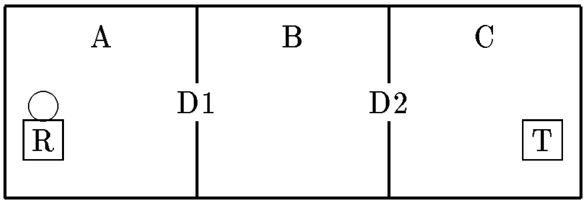

Figure 1: The robot's environment

to any object located in its current room.[1] The predicate `in` identifies the room in which the robot is currently located, and `next-to` identifies objects it is adjacent to.

It is clear that when planning in this domain, the robot's location within a particular room and the question of whether or not a door is open should be deferred while a high-level plan is being developed; the reason is that we can always expect to be able to develop a plan to either open a door or move the robot next to any object in its current room. The natural hierarchy in this domain is therefore one where we plan the `go-through` actions first; let us suppose that we next plan the `make-open` actions, and finally the `go-to` actions. It is well known that this hierarchical view of the problem results in an exponential reduction in the size of the search space [11], and this reduction is achieved by hierarchical planning systems such as SIPE [16] or ABSTRIPS [13] (not to mention many others).

Now suppose that our robot's environment is as shown in Figure 1, consisting of three rooms, `A`, `B` and `C`, connected by doors `D1` (between `A` and `B`) and `D2` (between `B` and `C`). The robot begins in room `A` with all of the doors shut and the goal of standing next to a table in room `C`. A hierarchical planner using the hierarchy described above would generate the plans shown in Table 2. (We have abbreviated `go-through` to `through` and `make-open` to `open` in the table.)

In existing hierarchical planners, this decomposition of the problem is achieved using a special operator representation. In ABSTRIPS, for example, each subgoal has an associated criticality value that indicates how long that subgoal should be deferred.

[1] The axiomatization in Table 1 fails to indicate that going to an object x should delete any existing `next-to` facts. This is an instance of the *ramification problem* [4], and can be treated in a variety of ways. The arguments we will present are independent of the treatment used.

Abstraction level	Plan	Truth value
1	`through(D1)`, `through(D2)`, `go-to(table)`	d^2t
2	`open(D1)`, `through(D1)`, `open(D2)`, `through(D2)`, `go-to(table)`	dt
3	`go-to(D1)`, `open(D1)`, `through(D1)`, `go-to(D2)`, `open(D2)`, `through(D2)`, `go-to(table)`	t

Table 2: Refining plans

2.2 Treatment using defaults

Base-level default knowledge can reproduce this hierarchical decomposition without resorting to the use of a specialized representation language. In the above example, we simply provide information indicating that certain subgoals are likely to be easily achieved:

$$\begin{array}{rl} \texttt{achievable}(\texttt{open}(x)) & [d^2t] \\ \texttt{achievable}(\texttt{next-to}(d)) & [dt] \end{array} \tag{1}$$

Our notation here is that dt corresponds to a first-order default, d^2t to a second-order default, and so on. The commonsense meaning of the first default in (1) is that we can open any door, and the second means that we can easily get to any object in our current room. The different default values of dt and d^2t are used to indicate that we are more certain of our ability to navigate around a particular room than to open a door.

To use the information in (1), suppose that the form of the axiom telling us when an action succeeds is as follows:

$$\texttt{occurs}(a) \wedge [\forall p . \texttt{prec}(p, a) \supset \texttt{sat}(p)] \supset \texttt{succeeds}(a)$$

$$\texttt{holds}(p) \supset \texttt{sat}(p) \tag{2}$$

Different declarative systems will obviously represent actions differently, but this seems a reasonable choice. An action succeeds if it occurs and its preconditions are satisfied; a specific precondition is satisfied if it holds in the situation in question. This description is the one actually used in [5, 7], with the modification that a modal operator is used to minimize the extent of the predicate `prec`.

We can now modify (2) to become the following:

$$\texttt{holds}(p) \vee \texttt{achievable}(p) \supset \texttt{sat}(p) \tag{3}$$

As we will see in Proposition 2.1, this has exactly the desired effect of capturing the hierarchy in our declarative setting.

Both of the defaults in (1) are base-level information about the planning domain and can now be used to control the planner's search. Specifically, the search for a plan that has truth value d^2t can essentially ignore the question of whether doors are open and where in the room a particular object is; the subsequent search for a refined plan with truth value dt will ignore only the question of where objects are within rooms. Somewhat more specifically, plan 1 in Table 2 has truth value d^2t, plan 2 has truth value dt and plan 3, which is guaranteed to succeed, has truth value t.

The two default statements cause the declarative planner to generate partial plans that are identical to the ones generated by a traditional hierarchical planner. We can also mechanize the process of turning an ABSTRIPS hierarchy into a set of defaults:

In ABSTRIPS, relations are assigned priorities; when solving the problem, the relations are considered in order of decreasing priority. In the example we are considering, the `open` predicate would be assigned a value of 2 and the `next-to` predicate a value of 1. In other words, the initial plan can ignore all instances of `open` or `next-to` as preconditions; the first refinement can ignore instances of `next-to`.

Proposition 2.1 *Suppose that an* ABSTRIPS *system labels each of its predicates* p_i *with a priority* v_i*. Then the functioning of* ABSTRIPS *can be duplicated using the defaults*

$$\texttt{achievable}(p_i) \quad [d^{v_i}t] \tag{4}$$

Proof. Suppose that the highest priority assigned by the system is m. Then at iteration k, ABSTRIPS will produce a plan that ignores all preconditions with priority less than or equal to $m-k$; we need to show that this is identical to the problem of finding a declarative solution to the planning problem that succeeds with truth value $d^{m-k}t$.

To see this, let a be any action that occurs as part of such a plan. We claim that a will succeed with truth value d^jt if and only if a succeeds while ignoring all preconditions of priority j or less; the proposition will then follow. But this is an immediate consequence of (3): If a succeeds when ignoring preconditions of priority j or less, since each of these preconditions is achievable with truth value d^jt, we can use (3) to prove (also with truth value d^jt) that the action succeeds. Conversely, if we can prove that the action succeeds with truth value d^jt or better, it must be the case that the only achievability assumptions used defaults with truth value d^it for $i \leq j$, so that we are ignoring only preconditions of priority j or less.

In making this argument, we are using the fact that there is no way to derive $\neg$`achievable`(p) from other information in our database, so all of the relevant achievability assumptions hold at any particular level of the default hierarchy. □

Proposition 2.1 shows that we can always continue to enjoy the computational benefits of hierarchical planning in a declarative setting.[2]

Deducing control information from base-level knowledge about the domain (default statements about preconditions) allows us to select our hierarchy automatically. SIPE and NONLIN, for instance, do not address the problem of classifying operator preconditions into filters and subgoals (Wilkins calls them *preconditions* and *subgoals* [16]). Filters are preconditions to an action which the planner should not, for heuristic reasons, plan to achieve. Subgoals are preconditions that the planner should try actively to achieve. The classification of preconditions into filters and subgoals in these classical planners is generally left for the user to solve in an ad hoc manner, using a hierarchy or in some other fashion.

[2]Note that we are assuming that it is possible to use our declarative system to draw default conclusions *without* performing the consistency check required by most formalizations of nonmonotonic reasoning. This is to be expected; the original arguments in [6] suggest that the computational benefits associated with nonmonotonicity are a result of the search focus provided by default conclusions when the consistency check is not performed. An implemented system that behaves in this fashion is discussed in [7].

Action	Preconditions	Effects
`go-through`(d)	`in`($r1$) `door`($d, r1, r2$) `open`(d) `next-to`(d)	$\neg$`in`($r1$) `in`($r2$)
`make-open`(d)	`in`($r1$) `next-to`(d) `door`($d, r1, r2$) `unlocked`(d)	`open`(d)
`go-to`(x)	`location`(x, r) `in`(r)	$\neg$`next-to`(y)
`unlock`(d)	`key`(k, d) `holding`(k) `next-to`(d)	`unlocked`(d)
`take`(k)	`cabinet`(k, c) `next-to`(c)	`holding`(k) $\neg$`cabinet`(k, c)

Table 3: Actions in the extended robot world

Given base-level default information about the achievability of certain predicates, the deferrable preconditions of an operator become precisely those for which achievability can be proven by default. The problem of distinguishing filters from subgoals thus becomes one of correctly stating the base-level assertions. Provided that the default assertions fairly represent our knowledge about the world, we expect that the induced hierarchy of actions will also be correct. Similar claims are made for the derivation of control knowledge generally in [8].

3 Beyond simple hierarchies

We now extend our example in a way that reveals some limitations of conventional hierarchical descriptions, and go on to show that default information can be used to avoid these limitations.

3.1 Extending the example

We complicate the world of the robot a bit, adding cabinets in every room and specifying that doors have locks on them. Inside the cabinets are keys that unlock the doors, although there is no guarantee that the cabinet in any particular room will contain the keys to that room's doors.

The actions in our extended domain appear in Table 3. We can unlock any door we are next to, if we have the key. We can take a key if it is in a cabinet that we are next to. The problem of opening a door d now becomes much more subtle. If d is locked, we need to find a path through unlocked doors to the cabinet that contains d's key.

3.2 The failure of static hierarchies

Consider now the problem of writing a set of hierarchical operators corresponding to the primitive actions available to our robot. It is clear that we must consider the type predicates `key` and `door` to be of the highest criticality, since they are unchanged by any actions we might perform. With a little thought, it becomes clear that the `in` precondition of the `go-through` operator clearly should also be planned for early.

What about the `open` precondition of the `go-through` operator? If we continue to let `open` have a lower priority than `in`, the openness of doors will only be considered after planning a complete route from start to finish. However, this ordering is likely to result in an abstract plan that we will be unable to expand. In expanding a plan that involves opening a locked door for which we have no key, the planner will generate a subplan for the robot to go to another room, retrieve the key and come back to the first locked door. This subplan will set off a cascade of sub-subplans for retrieving the keys to the locked doors that the robot runs into while on its first subtask and so on; if the planner succeeds in solving its problem this way, it will be by blind luck more than anything else. Any plan found is almost guaranteed to be disastrously baroque.

Note that this problem arises if the planner even *once* makes an incorrect assumption that a door will be easily opened. This is obviously quite likely for any plan that takes us through many doors.

3.3 Solutions in the static model

If we are committed to the conventional description of hierarchies, we can salvage it in one of two ways. We can abandon the hierarchy altogether, or split one of our operators into two.

3.3.1 Abandon

By giving `open` a high criticality value, we can choose to never defer it. Our abstract plans will no longer suffer from the "cascade" effect, but abandoning the hierarchy will cause us to lose the computational benefits of hierarchical planning. This seems not to be a viable solution for computational reasons, especially if most of the doors are unlocked.

3.3.2 Split

The other option is to split the `go-through` operator into two. One operator tells how to walk through easy doors; the other tells how to walk through hard doors. Assuming that our representation allows it, `go-through-easy-door` will give `open` a low criticality value, and will have filters making the rule applicable only if the door is unlocked or a key can easily be retrieved. `go-through-hard-door` will give `open` a high criticality value and will have filters guaranteeing that it is only applied when the door is in fact difficult to open. This will guarantee that we only defer `open` when it really is deferrable, avoiding the problem of Section 3.2. An alternative but equivalent split creates the operators `unlock-with-key`, which unlocks doors for which we have the key or can get it easily, and `unlock-without-key`, which deals with the more difficult situation.

But now suppose that we have another precondition to `go-through` that is also sometimes deferrable and sometimes not. Perhaps it is easy to get next to most doors, but for some `next-to` is a problem because there are obstacles obstructing the path to the door; moving the obstacles might require retrieving tools from other rooms. We now have the same problem with `next-to` that we did earlier with `open`: If we always defer `next-to`, a new cascade of subgoals makes it unlikely that we solve our original problem. If we never defer `next-to`, we lose the computational advantages of hierarchy. Sometimes deferring `next-to` forces us to split the `go-through` action into two again. Since we have already split this action along the easy-door/hard-door axis, the new split will result in four subactions. The following result is immediate:

Proposition 3.1 *If an action has n preconditions, each of which is deferrable in some situations but not in others, a static hierarchy may need to split the action into 2^n subtypes in order to plan effectively.* □

Note that it is not only the size of the domain description that is growing exponentially here; by introducing more actions we are also increasing the planner's branching factor from n to 2^n. We see from this that splitting the operators used by conventional hierarchical planners is not an attractive option in general.

3.4 Solution in the declarative model

In the declarative approach, solving this problem is trivial. If a predicate p_i should be given priority v_i *but only in the situation* s_i, we need merely replace (4) with

$$s_i \supset \texttt{achievable}(p_i) \quad [d^{v_i}t] \tag{5}$$

The s_i's appearing here will refer to various features of our domain; one instance of (5) might be

$$\texttt{holding}(k) \wedge \texttt{key}(k,d) \supset \texttt{achievable}(\texttt{unlocked}(d))$$

(with truth value d^jt for some j), saying that we don't have to worry about unlocking a door for which we already have the key. Given this axiom, note that we will assume that we can unlock d if there are other actions, also succeeding with truth value d^jt or better, that manage to establish s_i. So if we continue to make `next-to` the least critical of our preconditions, we can easily plan to collect keys and use them before worrying about the details of approaching the cabinets in which specific keys are contained.

Situation-dependent hierarchies arise in other settings as well. Consider the original axiomatization of our robot's world once again, where we indicated that opening a door d had three preconditions: that the robot be in the same room as d, that it be next to d, and that d be a door. The first of these preconditions is in some sense irrelevant, since the robot can't be next to d unless it is in the same room as d is.

Irrelevant though it may be from a declarative point of view, the precondition is crucial to a hierarchical planner working in this domain. To see why, recall that when constructing a plan that succeeds only by default, we ignore any precondition involving `next-to`. When we do this, the `make-open` action will have only a single precondition – that the object being opened be a door.

The result is that the robot, trying to get to the table in Figure 1, may form the plan of first opening both doors and only then walking to B, C and the table! By including the dummy `in` precondition to the `make-open` action, we ensure that the robot, when constructing the default plan, cannot expect to open a door d unless it is in the same room as d is.

The declarative approach provides us a much cleaner way of achieving this affect. Instead of saying that

$$\texttt{achievable}(\texttt{next-to}(d))$$

is true by default, we say that

$$\texttt{location}(d,r) \wedge \texttt{in}(r) \supset \texttt{achievable}(\texttt{next-to}(d)) \tag{6}$$

is true by default. This really captures the sense of the default: We can maneuver next to any object in our current room, but not necessarily next to other objects in the domain.

There are a variety of advantages to this formulation. First, existing methods require us to add a dummy `in` precondition to every action that has a `next-to` precondition; it is obviously simpler to merely replace (1) with (6).

Second, there are domains that are awkward to describe using the dummy preconditions needed by static hierarchies. Imagine that our domain contains some invisible objects, and the default we want to introduce says that we can only maneuver next to *visible* objects in our current room.

If we want to avoid splitting our actions into subtypes depending on the visibility of the door or other objects involved, we will need to introduce a dummy `visible` precondition. But this is unfaithful to the sense of the original domain description, which permitted us to open invisible doors if we could somehow manage to maneuver next to them.

These collected observations suggest another way of looking at Proposition 3.1. What we have seen is that the question of the criticality or priority to be assigned to a precondition is one that requires us to do some reasoning. How hard is it to open a door or get next to it? A static hierarchy such as that used by existing planning systems cannot do this kind of reasoning, and must determine the priorities by lookup alone.

Of course, there is a standard way to ensure that no reasoning is needed to solve a problem: Instead of working with the existing axiomatization, use its deductive closure. This guarantees that no inference will be needed in determining the priority levels, but the deductive closure of a database consisting of sentences like (5) can be expected to be of size exponential in the size of the original database. This is why the number of action subtypes grows exponentially with the number of split preconditions.

In some cases, of course, computing the deductive closure of a declarative database can save a great deal of effort at runtime; it might be very difficult to backward chain through rules like (5) to determine what priority should be assigned to a predicate in some particular situation. This seems unlikely to be true in planning examples, however, since the priority determination can be expected to depend in a fairly straightforward way on the values taken by situation-dependent variables (is the door locked?). In those cases where significant inference is needed to get from situational information to criticality values, we can still choose to forward-chain on the rules being used without splitting our actions into a catastrophic number of subtypes.

4 Conclusion

Lifschitz proves in [12] that any system making the STRIPS assumption will have difficulty dealing with the ramification problem. Wilkins acknowledges that such a system cannot be powerful enough to encode complex control knowledge: "Systems such as SIPE that do hierarchical planning can use abstract operators to encode some metaplanning knowledge, but ... the most interesting metaplanning ideas cannot be encoded in this manner" [15]. In this paper, we have confirmed Wilkins' statement, shown that the problem is closely related to the need to split STRIPS operators into subtypes to solve the ramification problem, and proposed a way to use base-level default information to overcome these difficulties. Atomic defaults can be used to capture existing hierarchical notions, while compound default sentences can be used to represent hierarchies that have previously eluded formulation.

Other authors have also considered these issues, although their work has a flavor rather different from ours. Feldman and Morris discuss the distinction between filters and subgoals in [2], but their analysis focuses on the detection of loops in planning search and leaves the question of hierarchy unaddressed. Knoblock discusses the automatic derivation of a planning hierarchy in [10], but the methods used are primarily syntactic and are concerned with the development of a static hierarchy in any event. Smith and Peot suggest that Knoblock's analysis should focus on more semantic features in [14] but they, too, limit their attention to static hierarchies.

This is to be expected; it seems impossible to incorporate dynamic hierarchies into nondeclarative planners. Although it is true that declarative planning has problems of its own, the analysis we have presented shows that it also has important capabilities that nondeclarative methods lack.

Acknowledgment

This work has been supported by the Air Force Office of Scientific Research under grant number 90-0363, by NSF under grant number IRI89-12188 and by DARPA/Rome Labs under grant number F30602-91-C-0036, and by an Air Force Graduate Laboratory Fellowship from Rome Labs. We would like to thank Don Geddis, Will Harvey and the other members of Stanford's PRINCIPIA group for helpful discussions on this work. Subbarao Kambhampati pointed out the relationship of Feldman and Morris's work to our own.

References

[1] A. B. Baker. Nonmonotonic reasoning in the framework of situation calculus. *Artificial Intelligence*, 49:5–23, 1991.

[2] R. Feldman and P. Morris. Admissible criteria for loop control in planning. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 151–157, 1990.

[3] R. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[4] J. J. Finger. *Exploiting Constraints in Design Synthesis.* PhD thesis, Stanford University, Stanford, CA, 1987.

[5] M. L. Ginsberg. Computational considerations in reasoning about action. In *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*, Boston, MA, 1991.

[6] M. L. Ginsberg. The computational value of nonmonotonic reasoning. In *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*, Boston, MA, 1991.

[7] M. L. Ginsberg. Modality: The key to practical knowledge representation. Technical report, Stanford University, 1992.

[8] M. L. Ginsberg and D. F. Geddis. Is there any need for domain-dependent control information? In *Proceedings of the Ninth National Conference on Artificial Intelligence*, 1991.

[9] M. L. Ginsberg and D. E. Smith. Reasoning about action I: A possible worlds approach. *Artificial Intelligence*, 35:165–195, 1988.

[10] C. A. Knoblock. Learning abstraction hierarchies for problem solving. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 923–928, 1990.

[11] R. E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26:35–77, 1985.

[12] V. Lifschitz. On the semantics of STRIPS. In *Proceedings of the 1986 Workshop on Planning and Reasoning about Action*, Timberline, Oregon, 1986.

[13] E. D. Sacerdoti. *A Structure for Plans and Behavior.* American Elsevier, New York, 1977.

[14] D. E. Smith and M. Peot. A critical look at Knoblock's hierarchy mechanism. Unpublished manuscript, 1992.

[15] D. E. Wilkins. Domain-independent planning: Representation and plan generation. *Artificial Intelligence*, 22:269–301, 1984.

[16] D. E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann, San Mateo, CA, 1988.