Before getting into the main topic—the state of the art in integrating conventional and reactive planning for use in intelligent agent planning—we will go into a little of the history of planning and its importance for artificial intelligence. Most of what we have to say here will be primarily of historical interest, but it will help the reader better understand the context of the state of the art of the field.

1.1 A Brief History of Planning

The first question to be answered is “why plan?”. The need for planning was recognized thousands of years ago. Sun Tzu, in his classic book *The Art of War*, written 2500 years ago, said: “The general who wins a battle makes many calculations in his temple before the battle is fought.” Some of the early successes in artificial intelligence were attained without the need for planning. For example, chess-playing programs and expert systems (Shortliffe, 1976 among many others) do not require any type of planning. The
answer, we feel, is that any intelligent agent capable of carrying out a sequence of actions is going to need to be able to plan. Chess-playing programs, while they may look several moves ahead as part of a heuristic search algorithm, do not actually plan more than one move ahead at a time. A simple expert system, after making all inferences possible based on available data, generally makes a single recommendation or diagnosis, and leaves the planning of a complex response up to the human agent who receives the recommendation.

We believe that such systems are quite limited in that they require human intervention after each step. When an agent has a more sophisticated ability to plan, it is capable of interacting with a human at any time the human would so desire, but it is no longer limited by the requirement for such interaction. For example, an expert system with the ability to plan would be able to recommend a plan of action based upon available data. That plan could include gathering new data at appropriate intervals, making further inferences, and doing further planning, based upon such new data.

The idea of planning started out from the (very wishful) notion that an agent can be given a complete description of the world in its initial state, a complete description of the world in its desired final (goal) state, a complete set of deterministic actions that can be performed, and from there can determine the most efficient set of actions to transform the initial state into the goal state. For example, the General Problem Solver (GPS) of Green (Green, 1969) approaches planning as a theorem-proving activity, which is a completely deterministic domain (although computationally intractable). GPS also uses means-ends analysis—simultaneously working forward from the initial state and backward from the goal state—to reduce computational complexity.

Many of the planners that will be described in this introductory section can be divided into two large classes called conventional planners and reactive planners. We define the difference between a conventional and a reactive planner as follows:

**Definition 1.1**

A *conventional planner* is one that establishes—given an initial state and a goal state—a deterministic sequence of actions for getting from the initial state to the goal state.

A *reactive planner* enumerates—for each of as large a set of states as possible—an action for responding to that state should it ever arise. The outcome of that action may not be completely known in advance.

Most planners, however, do not fit neatly into one or the other category, but rather can be seen as lying on a continuum of sorts as shown in Figure
1.1 A Brief History of Planning

1.1. This figure shows some of the different characteristics of the two types of planning systems. For example, reactive planners tend to be more geared toward responding to emergencies rather than planning out a sequence of actions geared toward achieving a particular goal. The effects of actions tend to be less predictable in a reactive planner than in a conventional planner. When one is less goal-directed, it matters less that an action will have a particular effect than simply that its value be expected to be greater than for other action, or than doing nothing. Reactive planners tend to be capable of constructing plans that respond to a large number of states, as opposed to conventional planners, which usually only deal with states directly on the path from the initial state to the goal node. However, inasmuch as a given planner will exhibit some of the characteristics of each type of planner to varying degrees, it is possible to classify planners roughly in a continuum from “reactive” to “conventional.”

![Figure 1.1 Reactive vs. Conventional Planners](image)

**Figure 1.1 Reactive vs. Conventional Planners**

The importance of this general classification may not be completely clear at the outset, but let us say that the main theme of the book is addressing the problem of when to use a conventional planner, when to use a reactive planner, and when not to plan at all—often when the decision itself is something that must be made in real-time under real time constraints. Thus, within this larger, more robust framework that we hope to build for the reader, conventional and reactive planners are the basic building blocks, and it helps to have a clear understanding of what each type of planner is. We will start here with a historical look at these two types of planners.

Conventional planners tended to be very formal and mathematical in their general approach. That is, planning was seen as a mathematical problem, a formal mathematical solution to that problem. To the extent that a problem could be characterized in a completely formal way, therefore, conventional
planning worked very well. A conventional planner would take as inputs the initial state, the goal state, and a set of actions for moving from one state to another. It would look for a sequence of actions which, when applied one after the other to the initial state, would produce the goal state. In a very general sense, it did this recursively. In the simplest form, the planner would look at all possible actions he or she could perform, starting with the initial state. If one of the actions led to the goal state directly, the planning process was done. If not, then the planning problem could be replaced by the problem of getting from one of the new states, obtained after performing one action, to the original goal state. If any one of these problems could be solved, a solution existed to the planning problem. The planner would recursively try to solve each of these derived planning problems in turn.

In other words, conventional planners attempted to find the path from the initial to the goal state by using a search algorithm, and as such they can be viewed as special cases of search algorithms. Finding a good search algorithm has been a large problem that has interested artificial intelligence (AI) researchers since the inception of the field. For example, a chess-playing program will search all possible moves, then all possible countermoves by the opponent, and so on. It looks as many moves ahead as possible to determine the next best possible move for the player. A conventional planner will do something similar, in that each action which it has to work with is something like a move in a game of chess. Each move must be formally specified, and there is an initial state and a set of goal states (checkmate positions). Conventional planning differs from chess in the sense that the planner gets to select all moves, whereas a chess program gets to choose only alternate moves (the opponent chooses the intervening moves).

However, a conventional planner differs from a good chess-playing program in a couple of other important ways. For the planner to be any good, he or she must enumerate a complete set of actions to get from the initial state to the goal state, whereas a chess-playing program does not need to know precisely how to get to checkmate in order to make a move. Also, perfection is asked of a conventional planner, whereas the chess-playing program need only outwit its opponent.

All good search algorithms employ some kind of heuristic to help them search through the space of possible actions. In the case of a chess program, a heuristic might help it to evaluate a position—for example evaluating a position where one’s opponent’s queen is threatened very favorably. Such heuristics not only allow the search program to make a final decision on what move to make, but they also reduce search complexity by enabling one to prune the search space. For example, alpha-beta pruning allows the search
1.1 A Brief History of Planning

program—once it has seen that the opponent has a response to a particular move that results in a less desirable position than anything that could happen by taking a different move—to discard that move as undesirable without looking further at other opponent responses. Conventional planners attempted to apply similar heuristics to the planning process. In general, any action that gets one closer to achieving the goal state is rated highly by the heuristic in the planner. For example, if the goal requires achieving a number of subgoals, then achieving just one of the subgoals will be seen as heuristically desirable. Less complex actions might also be seen as more desirable than more complex actions.

Conventional planners suffered from a number of significant limitations. The first stemmed from the fact that the problem had to be stated completely formally. If it was not possible to describe precisely every possible state the planner could get into, as well as the precise effects—and lack of effects—of every possible action, then the planner couldn’t even be invoked, much less solve the problem. Most of the time, in the real world, the effects of actions are not known in advance completely—conventional planning begins to break down as uncertainty increases. Moreover, the state of the world is not always known completely at all times—in fact it usually is not known. Sometimes you have to begin taking action even when you don’t know everything about the world that might be relevant to your decision. Even in a completely formal sense, there was something known as the frame problem, where it is not always easy to enumerate just what the side effects of specific actions are.

Conventional planners can also be very fragile in their general approach. If for any reason the execution of a plan failed, there was no notion of recovery, at least in the most traditional planners. The execution of the plan assumed that each step simply proceeded without checking whether the previous step had been properly executed. Again, in practice sometimes plan execution will fail: the effects of an action will be different than previously expected, or an action simply won’t be taken when it should for whatever reason, or the world changes unexpectedly. All these factors would simply put a conventional planner out of commission.

Another weakness stemmed from the fact that the original conventional planners did not really plan. They simply, as alluded to above, searched. Though a search may be seen as a special case of planning (and from another point of view, planning may be seen as a special case of search), search is limited in that it is always operating at the lowest level of granularity. If one is planning a trip, then usually the lowest level of detail—such as where to flag a taxi for the ride to the airport—is not looked at first. More general planning,
like the cities to be visited, is done first. The details must be filled in at some point. But if you are forced to look at the details too early, because of interactions between different low-level details in different parts of the plan, then the search problem created is intractable, simply because there are so many details at the lowest level.

A related problem is that the traditional conventional planner is domain-independent. That is, planning is viewed as a general purpose, black-box program into which just about any planning problem can be inserted. The problem statement includes a formal description and a domain description; a suitable plan is generated from this statement. Even if this worked well (and the preceding paragraphs indicate some limitations), it becomes difficult to see how to encode heuristics in a purely domain-independent fashion. As mentioned above, a few very general-purpose heuristics (like going for a partial solution) seem plausible, but it is difficult to see how domain-independent heuristics can possibly be as powerful as a domain-dependent planner that had a wealth of domain-specific heuristics.

The earliest conventional planners consisted of such systems as the aforementioned GPS and STRIPS (Fikes, Hart, Nilsson, 1972). Once the problems cited in the last few paragraphs had been noticed with the earliest conventional planners, a number of concurrent steps were taken in a partial attempt to rectify these problems. These new planners, which were really still just conventional planners but with added features, might be classified as nonlinear planners, hierarchical planners, constraint-based planners, and plan-monitoring and execution planners.

One of the assumptions made by the original conventional planners was that planning involved designing a sequence of actions for getting from an initial state to a goal state. At all times during the planning process, therefore, there was a sequence of actions. Nonlinear planners relaxed this assumption. In the case of a nonlinear planner, there could be multiple branches in the plan that were not assumed to be ordered in any way. Nonlinear planning tends to proceed in a natural way, given the goal state. Each subgoal that was part of the goal gave rise to a separate branch in the now nonlinear plan. As the planning process proceeds, and intermediate goals need to be achieved at various points, these too could give rise to different branches in the plan. The idea of this nonlinearity is twofold. One, it enables the planner to avoid having to commit too early to a particular order among actions. Second, it reduces computational complexity by reducing planning into a number of more manageable parts. Nonlinear planners included such examples as NONLIN (Tate, 1977) and NOAH (Sacerdoti, 1977).
However, nonlinear planning itself had problems, in that there could be a conflict between the different branches of a plan. Achieving one subgoal may have side effects that thwart the attempt to achieve other subgoals. In general, the way in which nonlinear planning proceeded was to first assume that there were no such conflicts. Then a set of special programs called critics would be invoked whose job it was to notice such conflicts and provide a resolution. Generally, the most prevalent item in the critics’ bag of tricks was to impose some kind of order on the different subgoals. Thus, conflict could be avoided if only one critic performed one subgoal before the other, or in some cases if one tried some more ingenious interleaving.

The critics almost by definition had to be domain-independent, a disadvantage. Moreover, it was far from clear that these planners were provably correct. Everything depended on the critics, and if the critics could not be counted on both to notice all potential conflicts and to correctly resolve them, then the planning process would fail. There was little reason for confidence, especially since critics tended to be black-box programs.

Hierarchical planning represented another evolution in the planning paradigm. Hierarchical planning is sometimes confused with nonlinear planning, and indeed there were a number of examples of planning systems that did both kinds of planning. In hierarchical planning, an abstract plan is first constructed to solve an idealized version of the real problem. This abstract plan is then mapped, using some well-understood mapping, into the “real world.” The plan as first mapped is generally not a correct solution to the original problem, so some replanning is generally necessary to rectify this situation. Often there are several different abstraction levels, with the planning process gradually proceeding from more idealized worlds, to less idealized, down to the real world.

Just how these idealized worlds are constructed from the real one will have an important impact on the performance of the planner. One simple way in which this was done in one planning system was to ensure all actions have certain preconditions. Then some of those preconditions were simply ignored in the more idealized worlds. As the world model became more realistic, more preconditions were introduced and previously valid plans might have needed to be revised or made more complex. Another type of hierarchical plan might involve the use of abstract actions. In the more realistic world, each abstract action would need to be resolved into a series of actions, and this would require a replanning process.

The question of whether a planner is hierarchical is orthogonal to whether it is nonlinear. A planner could be hierarchical in the sense that the planner proceeds from an idealized world to the real world, and yet in all these worlds
it may have a completely linear plan. Alternatively, it could have only a partial order on its actions at all times, and yet all actions could take place within the single world. Or a planner could be both hierarchical and nonlinear.

The biggest advantage gained by using a hierarchical planner is the reduction in computational complexity. The initial plan generated at the first planning stage is generally simpler than that generated later in the process, and therefore will require less time to construct. Then, in later stages, computational complexity is reduced because the planner is revising an existing plan rather than constructing one from scratch. With regard to disadvantages of hierarchical planning, the biggest seems to be that there is no guarantee of completeness. It is possible that the first step in the planning process will produce a plan that cannot be refined into a workable complete plan, and the hierarchical planning paradigm makes no obvious allowance for this type of possibility. Examples of hierarchical planners were ABSTRIPS (Sacerdoti, 1974) and NOAH (Sacerdoti, 1977).

One weakness of all types of planners mentioned so far is that they assume a world with unlimited resources. They may attempt to construct a plan that is optimally efficient, but they do not have any notion that if certain resource limitations are exceeded, the plan may be completely unusable. The next group of planners, which might be described as constraint-based planners, attempted to overcome this limitation. Such planners included SIPE (Wilkins, 1984) and DEVISER (Vere, 1981).

There are many different types of resource constraints that a plan may have. A plan may have an overall limit on the amount of time it is allowed to take, and each action in the plan may take a certain amount of time. Nonlinearity could then be used to perform certain tasks in parallel. However, there may also be constraints on which tasks can be performed in parallel—each action may use certain resources for their duration, and there may be only a limited number of such resources. These resources would not be consumed in the sense that they would become available again once the step was completed, but this could put constraints on the total time taken by the plan. Other resources may be of the type that are consumed, and there may be only a certain quota allowed at the beginning of the plan.

In general, the concept of nonlinear planning applied well to constraint-based planning. As noted earlier, in nonlinear planning critics are used to prevent conflicts on different parallel branches of a plan. Constraint-based planning requires additional critics to take into account the different constraints, but the general approach to planning is similar.

Another general concept in domain-independent planning was that of refinement planning. Refinement planning is based upon the idea that one
A Brief History of Planning

starts from considering all possible plans as potential solutions to the particular planning problem one is faced with; one then introduces step-by-step refinements to that set. Each refinement reduces the set of possible plans until one is left with just the actual plans that solve the problem. Refinement planning is in some sense just another way to think of planning, because most of the approaches to conventional planning discussed so far can be thought of as refinement planners. For example, hierarchical planning involves restricting the set of possible plans to all plans that are an extension of the existing high-level plan at a certain point in the planning process. Nonlinear planning involves gradually introducing constraints on the order in which actions in the plan are performed, which involves a similar process of restricting the set of possible plans.

But by understanding planning in refinement terms, it does provide other means for looking at the planning process by thinking in terms of the specific type of refinement used by the planner. For example, forward state-space refinement—(Drummond, 1989) and (Bacchus & Kabanza, 1995)—involves gradually expanding the partial plan starting with the initial state, and only adding steps to the end of the existing partial plan. Conversely, backward state-space refinement involves working backward from the goal state. Means-ends analysis is a form of forward state space refinement that allows consideration of only those actions that are somehow relevant to the top level goals.

Another type of refinement is called plan-space refinement. The distinction between state-space and plan-space refinement is that in state-space planning new actions are fairly tightly constrained into a particular part of the plan when they are introduced into the plan. In plan-space refinement, new actions may be introduced into the plan without any clear idea where they will actually fit into the plan. Thus, plan-space refinement can probably be understood as a very least-commitment type of planning. However, the danger in plan-space refinement is that the planning process itself is not very well planned. A variety of actions are gradually thrown into the soup of the developing plan, and the hope is that somehow they will be organized into a plan that actually solves the problem. This is done in part by introducing the idea of “clobberers” and “white knights.” A “clobberer” is an action that causes a precondition, required at a later state in the plan, to be untrue. A “white knight,” conversely, reasserts that the precondition is true. By requiring a white knight for every clobberer, the hope is that the planning can be completed in this rather unstructured fashion.

Plan-space refinement can be shown to have some interesting properties about its correctness (any plan it finds does solve the planning problem) and completeness (it will always find a plan if one exists), but unfortunately com-
putation complexity suffers. Thus, plan-space refinement, while perhaps providing another interesting dimension on which to view the planning process, is not really useful for a practical planner. Indeed, the canonical plan-space refinement algorithm TWEAK, uses what is known as a “nondeterministic” algorithm. A nondeterministic algorithm differs from a deterministic algorithm in that there is no control mechanism to decide what to do next; with a nondeterministic algorithm it is known only that following one particular branch of the tree will lead to a solution. To actually decide which path to explore would potentially involve exploring all of them, and backtracking until one found a path that worked. Generally speaking, this leads to a computational combinatorial explosion.

At the beginning of this section we observed that a high-level distinction can be drawn between conventional planning and reactive planning. All the planning paradigms that have been discussed so far may be seen as conventional planners. Figure 1.1 outlined some of the perceived differences between conventional and reactive planners, but perhaps the biggest difference is that a reactive planner assumes some intelligent presence in real time, whereas a conventional planner tends to hope that it can do its planning and then go home—hoping that the plan can be executed without further monitoring.

### 1.1.1 Reactive Planning

Before proceeding further into reactive planning, some discussion of terminology is appropriate. The reader may be understandably confused if we are using the term *reactive* differently than it is used either in other areas of AI literature, or in popular usage. As we use the term, *reactive planning* still takes place before execution time, when there is a (relatively) large amount of time available to construct plans and (relatively) few distractions arising to interfere with that process. Reactive planning, therefore, takes place at the same time as conventional planning. One difference, however, is that reactive planning presupposes the existence of some execution-time monitoring support, and its plan is constructed so as to take this into account. Thus, for example, when in Figure 1.1 we observed that reactive planners construct plans that are able to respond to a large number of world states, it was implicit that the reason reactive planners are able to do this is that there will be some type of monitoring of plan execution, and this monitoring will keep track of the actual world state at all times and be able to respond accordingly. Without this monitoring, a plan capable of responding to multiple states would be useless.
1.1 A Brief History of Planning

Reactive planning is also distinct, but goes hand in hand, with reaction. In our view, reactive planning is what happens before plan execution, but reaction is what happens at execution time. Reactive planning and reaction therefore go hand in hand but refer to different stages in the process.

In popular usage, it is generally considered wise to be proactive and a sign of weakness to be merely reactive. The person, or the intelligent agent, who is merely able to respond to events as they happen is generally seen as being at a disadvantage to the person who is able to anticipate and avoid the unpleasant consequences of undesired events. How do we justify promoting reaction when there is an understandable preference for proaction? The answer lies in the fact that we are interested in reactive planning as opposed to merely reaction. No matter how well prepared we are, sometimes there are going to be unpredictable events that arise. The process of reactive planning represents an attempt to be as well prepared for these unpredictable events as possible, so that their negative consequences are minimized. Thus, reactive planning is really a form of proaction, and for that matter conventional planning is also a form of proaction.

This book is fundamentally about the decision of whether to plan, and if so, what specific planning algorithm to use. Thus, in a superficial sense, there would be no reason to favor reactive planning over conventional planning or vice versa; one would simply decide which, if either, to use in a given situation, and having decided to use reactive or conventional planning, the more specific planning systems can then be selected. In one sense, this point of view is accurate. Yet at the same time, both authors come from more of a reactive planning background, and there is another sense in which it is really reactive planning that opens up the door to the “decision to plan.” This is so because in designing a reactive planner, one is almost forced to decide how much planning to do. A conventional planner only constructs plans that respond to a single state, so the decision of how much planning to do has already been made. But the plans produced by a reactive planner are capable of responding to a large number of states, and so the question arises of just how many states they should respond to. Clearly, one possible answer would be “all of them,” and indeed one reactive planning paradigm that will be explored, universal planning, does answer the question in that way. But there are also other answers that could be provided.

Reactive planning, therefore, forces one to make the decision of how much planning to do. The existence of a large number of reactive planning algorithms requires that the main topic of this book be addressed before they can be used in a meaningful fashion.
Reactive planning arose because of the limitations in conventional planning that were alluded to earlier. It was felt that conventional planning was very weak because the plans constructed were very rigid and hence brittle in a dynamic environment. All reactive planners represented an attempt to ameliorate that by planning in advance for problems that could arise in the real, dynamic world. Conventional planners also have no real notion of operating under a deadline, or of the fact that a deadline may be “hard” (any situation in which the deadline is not met is equally bad, even if the deadline is only missed by a small margin) or “soft” (utility begins to degrade at the deadline but does not drop to zero immediately). A reactive planner can take deadlines into account by planning to do things differently based upon how close a deadline is. For that matter, conventional planners had no notion of utility at all. From their point of view, a plan either solves the problem it is supposed to solve or it does not; the idea that some solutions may be better than others did not really enter into the planning process as it can with a reactive planner.

It could be argued that some reactive planners do not so much plan as simply provide an agent capable of reaction. We will attempt to show that most of what we are calling reactive planners do in fact plan for specific circumstances in the world, and that the “planners,” which are simply reaction agents, tend to be very undirected in their approach.

1.1.2 Types of Reactive Planning

Reactive planning, in turn, can be divided into a number of smaller areas, including planning for plan monitoring, hard-wired reaction, anytime algorithms, universal planning, decision theoretic analysis, and real-time planning. The last two categories, decision theoretic analysis and real-time planning, are actually hybrids of reactive planning and reaction, and will be discussed later.

1.1.2.1 Plan Monitoring

The oldest of these approaches is planning for plan monitoring, the main example of which is triangle tables (Fikes, Hart, Nilsson, 1972). In this paradigm, a conventional planner was still used to construct a plan. However, it was then conceded that this plan was brittle. The idea was to augment the brittle, conventional plan with monitoring information whose purpose was to make sure that the plan was being executed properly. Thus, a conventional planner tends to assume that the world will be in a certain state at a certain
step in the plan. The monitoring information would make explicit the assumptions that otherwise were merely implicit, and if these assumptions broke down, then something would happen.

That last sentence is intentionally vague, and its vagueness represents one of the biggest weaknesses of this approach. If the assumptions broke down, “something” would go wrong with the plan, and “something” needed to be done about it, but this approach leaves unanswered the question of just what that “something” should be. Presumably one thing that could be done would be to determine the new state of the world, and the original goal, and solve this new problem using the same planner as was originally used. This has two big weaknesses, though. One is that the new planning problem may not be as difficult as the original problem, and yet using this approach, solving it will be just as hard. The problem in plan execution might be correctable with just a single action, not requiring massive replanning. Furthermore, when something goes wrong at execution time, immediate action is probably required, and the ability to respond on deadline is not captured here at all.

A more subtle weakness is that this approach really requires three separate components at the plan execution stage: a plan executor, a plan monitor, and a replanner. The executor is responsible for actually taking action, the monitor checks to see whether the required assumptions are met and invokes either the next cycle in the plan executor or the replanner if the assumptions are not met. The weakness in the original approach was that these components weren’t really provided, but it should be fairly easy to see where the beginnings of a real-time architecture could be developed here, and how important it is.

1.1.2.2 Hard-wired reaction

Hard-wired reaction represented an attempt to develop a real-time architecture. Such systems included action nets (Nilsson, 1988), situated automata (Kaelbling, 1987), and subsumption architecture (Brooks, 1986). The idea here was to make the reaction process automatic by either encoding it in the hardware itself, or by providing software with provably correct properties involving its ability to respond to real-world events. Generally speaking, but not always, each of these planners provided some kind of language for encoding the behavior that was wanted at execution time. The programmer then became involved in the planning process by programming in this language. The reactive planning system included an algorithm for going from the language to the actual agent, which as mentioned could be hardware or software. These systems sometimes provided a fixed cycle time, to be able to guarantee particular results. If one can prove that in all possible world situa-
tions, the desired reaction can be provided in at most one millisecond after response begins, then a cycle time of two milliseconds might be used. The time is padded to be sure that response takes place before the end of the cycle. With such a system, response is then guaranteed within four milliseconds of the event happening (up to two milliseconds to complete the cycle in process when the event occurs, and up to two milliseconds to act in the next cycle). This is somewhat longer than the original one millisecond, but it is guaranteed, and the whole point of this type of argument was to be able to make such guarantees.

The strengths of this approach include the ability to guarantee a response within a particular period of time, which in any real-time domain is desirable. Other reactive approaches do not necessarily provide guaranteed response, of any type. The biggest weakness is implied by the name: the response needs to be hard-coded, generally by a human programmer. Thus, much of the advantage of the planning process is lost—namely, the idea is lost that the intelligent agent is doing the planning on its own. Still, the hard-wired approach represented another interesting step in the evolution of an architecture for reactive systems. There is no plan monitor to speak of, but there is a plan executor, and another notion, the idea of an execution cycle, is introduced.

1.1.2.3 Anytime Algorithms

The idea behind an anytime algorithm (Dean & Boddy, 1988) is that whereas a conventional algorithm must run to completion before producing a result, an anytime algorithm has a result available at all times during execution, so that if it runs out of time to continue execution, it will have a response available nevertheless. The question arises of what the connection is between an anytime algorithm and a reactive planner. The answer is that an anytime algorithm tends to be very good at dealing with a dynamic world. Whenever it is expected to respond, it is able to do so. So the design of an anytime algorithm can be seen as a form of reactive planning.

A simple example may help to illustrate the principle of anytime algorithms and their connections to reactive planning. Let’s say someone is planning a business trip and has only a limited amount of time to do the planning. An anytime algorithm for this problem would be to call up a series of airlines and ask each one for a price quote. The “anytime” response, which is available at all stages in executing this algorithm, is to use the best price quoted to date. If at any point it becomes necessary to curtail the search for the best price, there will be a viable alternative that can be used.

The anytime algorithm does not do reactive planning; rather, it is a form of reactive plan. The process of designing an anytime algorithm, however, can
be seen as reactive planning. For example, in planning the problem solution described above, such factors as which airlines to call first based on which tend to have the better price, or whether to use a travel agent, could be taken into account in the planning process. A more sophisticated form of reactive planning that accompanies anytime algorithms is to imagine that at all times an intelligent agent has available to it a number of anytime algorithms that can be executed. It can choose to devote time to one of these algorithms each cycle. The question is how to divide its time among the anytime algorithms. Here a notion of *diminishing returns* is used. The assumption is made that the marginal value of any anytime algorithm over time tends to decrease, so that it is always optimal to spend the next cycle on the algorithm with the current highest marginal value. Of course, this assumption is not always valid; among other things, the value of an anytime algorithm tends to grow as a step function, not continuously.

Anytime algorithms have their limitations: in most cases an anytime algorithm will be part of a larger reactive plan with other components, not just a reactive plan by itself. Still, for certain domains, they can be quite effective. Most chess programs, for example, can be seen as a form of anytime algorithm. Also, certain limited trip planning domains, such as the one outlined above, may yield to a pure anytime algorithm. Anytime algorithms also tend to be good at taking into account the notion of utility or value of action—they have to have some notion to be able to always improve their actions over time—and this notion will prove to be valuable for other reactive plans. Also, when the agent is assumed to have several anytime algorithms it can work on at all times, then once again some type of intelligent agent architecture is indicated.

1.1.2.4 Universal Planning

The idea behind universal planning was to plan out in advance responses to all events that might occur. Thus, it represented the most extreme possible interpretation of the reactive planning idea that one tries to respond to a large number of states. Of course, it would be computationally intractable, in terms of space requirements, to encode all possible states individually; the idea behind universal planning was that there be some way of encoding all these states through some categorization scheme. Even so, universal planning is believed to be computationally intractable. Universal planning is primarily the brainchild of Marcel Schoppers (Schoppers, 1987).
1.1.2.5 Decision Theoretic Analysis

Perhaps more interesting is the idea of decision theoretic analysis in the planning process. The basic idea here is that one sees the planning process as essentially a mathematical problem of maximizing the utility of the action(s) that are performed. Most decision theoretic analyses seem to assume that the mathematical properties of the various planning algorithms are well-understood, and seem to focus on the meta-problem of which planning algorithm to use on which problem. As such, they are probably not much good in designing real planners, although the general issues that they tackle are likely to be interesting ones in many domains. They also look at the meta-meta-problem of when to spend time working on the meta-problem and when to spend time working on the problem itself. Another idea that the decision theoretic algorithms looked at was the idea of bounded rationality. Bounded rationality refers to the idea that if you look at the value of “rational” thought devoted to a problem as a function of the amount of time spent thinking about it, it tends to be an asymptotic function with a definite upper bound. Doubling the amount of computational time devoted to a problem may result in only a small increase in the value of the result. In this sense, the notions of bounded rationality and decision theoretic analysis are closely connected to the idea of anytime algorithms talked about earlier in this section.

Decision theoretic analyses fall partly in the realm of reactive planners and partly in the realm of just doing reaction without planning. The assumption underlying all of this decision theoretic analysis is that it might be done at planning time or it might be done at execution time; indeed there is generally no precise distinction made between the two. To the extent that it is done at planning time, it can be seen as a reactive planner, and to the extent it is done at execution time it is a form of reaction. The idea is that a plan is constructed in advance, and as the world changes at execution time not only does the agent respond to those changes, but it also makes an effective decision about when to replan and when to simply continue with the existing plan. Examples of decision theoretic analyses include those by Horvitz (Horvitz, 1987) and Russell and Wefald (Russell & Wefald, 1991).

The primary difficulty with decision theoretic analysis is that it is just too sophisticated for use in real-world problems given the current state of the art. Given that there is to date not even agreement on what constitutes a good planner, too much focus on meta-problems and meta-meta-problems is not likely to be useful right now. Still, it represents an interesting way to look at these issues.
1.1.2.6 Real-Time Planning

The final, and probably the most practical, approach is known as real-time planning. A real-time planner generally has a well-defined architecture that includes some notion of a computational cycle and some idea of how long a computational cycle is intended to take. Planning is therefore interleaved with reaction depending on available cycles. If a large number of cycles are needed for reaction to immediate problems, then less time can be spent on planning. If there is more time available, or if situations that arise indicate the need for further planning, then the planning process can kick in again at any time. In addition to the notion of a computational cycle, there is usually some means of monitoring the world at execution time so that changes in the world can trigger the right response(s). Real-time planners are also generally capable of constructing partial plans as might be possible, and then refining that plan as more information becomes available. They are also capable of responding effectively to deadlines. All of this is made possible primarily through the use of the architecture that can make sure, in a controlled fashion, that everything the planner needs to take care of is being dealt with. Some real-time planners or planning architectures include those of Hendler (Hendler, 1990), Hayes-Roth (Hayes-Roth, 1985), and Washington (Washington & Hayes-Roth, 1989).

1.1.3 Summary

As mentioned earlier, we are interested here mainly in the question of when to plan and when not to plan. To a large extent, the planners discussed in this opening section represent candidate planners that can be used when the decision to plan is made. However, the last two types of planning—decision theoretic planning and real-time planning—do actually make this type of decision. Therefore, there will be considerable focus on these two types of planners in this book. This will not be to the exclusion of other planners, because as noted, those planners can solve particular problems even though they cannot make meta-level decisions.

There will also be a larger focus on reactive planning than on conventional planning. The reason is that generally if you are using a conventional planner, it implies a fairly simple planning problem, and one where the decision to plan has already been made in the affirmative. If the domain is sophisticated enough to require a nontrivial decision, then it is probably sophisticated enough to require a reactive planner.
As can probably be seen from the wide range of approaches to planning that have been used in about thirty years of AI research, planning is a difficult problem. It is difficult, at least in part, because planners must plan for dynamic, unpredictable, and very complex domains. Nevertheless, we believe both the state of the theory of planning research and the state of real-world technology have reached a point where much more sophisticated planning systems can be deployed in the real world, and it is our hope that we will be able to inspire readers to build such systems.

1.2 Types of Domains Used In AI

To test out any new concept in artificial intelligence, it is necessary to have a domain of application to be used as a testbed in order to validate the idea. The world of real-time planning is no exception. In this section, we give a brief summary of the different types of domains that have been used to validate artificial intelligence systems in general. We then discuss at a more detailed level how these different domains are suitable applications for real-time ideas. Finally, we will introduce a particular domain, flight planning, which will be used to illustrate the ideas presented in the remainder of this book.

First, though, a few words on what we mean by a domain is probably appropriate. It is a central tenet of most work in artificial intelligence that ideas, methodologies, or technologies that are developed will have general applicability, and will not just be narrow solutions to a single problem. Thus, for example, if an expert system is designed for medical diagnosis, then out of that some expert-system shell can be extracted that can then be used for other problems for which an expert system is a desirable solution. Or, if a planning system is designed for scheduling tasks in a manufacturing setting, a domain-independent planner can be distilled from that which would then carry over into flight planning.

By a domain we mean: A set of closely related problems such that the solution to one is likely to provide guidance as to the appropriate ways to solve others, and such that a human being who is expert in some of the problems is likely to have the ability to solve all problems in the domain.

This definition is necessarily a bit imprecise, so perhaps a few examples might be appropriate. Approving a credit card transaction is an example of a domain. It is a set of problems, not just a single problem: each new transaction that comes up for approval represents a new problem that requires a solution. But as one looks at the decisions made for different credit card transactions, gradually a pattern will emerge that will make it easier for the
decision to be made on future transactions. A human being who knows how to approve some transactions will likely have some skill in approving all types of credit card transactions.

However, the approval of credit card transactions is not the same domain as, say, the monitoring of a patient in a surgical intensive care unit (SICU). Similar real-time technologies might apply to both domains—they may both require response within real time—but specific knowledge about credit card transactions is not likely to help one in understanding the problems of a patient in the SICU. Of course, there are points when what constitutes a domain becomes a bit fuzzy. Is Visa credit card approval the same domain as American Express credit card approval? It is not necessarily trivial to decide in all cases what constitutes a domain, but certainly there are cases where different problems clearly fall into different domains.

It might be asked why we feel it is important to draw this distinction between domains, and put effort into defining when two problems fall into different domains as opposed to the same domain. The primary reason for this is: we are attempting to show that the ideas presented in this book are domain independent. One good way to show that an idea is domain independent is to apply it, or show how it can be applied, to a multitude of different domains. Certain types of domains, however, are definitely better for applying the ideas of this book than others, and so another purpose of this discussion of domain is to help illustrate which types of domain are best suited for these ideas.

The domains to which artificial intelligence ideas are applied could probably be categorized in almost as many ways as there are people working in the field, but a couple of subdivisions will prove to be interesting for the purposes of this book. One way to classify AI domains would be into toy, knowledge-intensive, and compute intensive. Toy domains refer to any domain that is constructed solely for the purpose of illustrating a particular AI solution. They tend to be quite simple in scope, and are generally rich only in the area that the solution is intended for, because the creator of the solution and the domain is primarily trying to illustrate the strengths of his or her solution. The biggest advantage in using toy domains is that it is possible to design experiments very clearly which test out hypotheses about the solution. Because the domain is designed only to test out the solution, less effort needs to be put into factoring out the effects of elements to the domain that are unconnected to the solution or where the solution is ineffective in solving the problem. Another advantage is that it allows for the effective comparison of different solutions presented by different researchers. If they attempt to apply their solution to real-world problems, then each researcher will likely
be using a different domain expert, so it becomes difficult to factor out the
effects of the domain expert on the solution. Yet another advantage is that it
may take time for the domain expert who knows about the real-world solu-
tion to come up with useful information about the domain in a form that the
AI expert can use, and using a toy domain in the interim allows for effective
early feedback about the value of the solution. On the other hand, the biggest
drawback in using a toy domain is that it tends to lead to “solutions in search
of a problem.” If the only problem(s) that can be found which the solution
solves are those that are explicitly created for the purposes of illustrating the
solution’s strengths, then it calls into question whether the technology solves
anything useful at all. At best it represents a partial solution to real-world
problems. Another disadvantage is that the toy domain may be tailored to
make the proposed solution look better than it really is.

The second major type of domain we wish to explore is knowledge-inten-
sive domains. These are “real-world” domains where a great deal of human
knowledge is required from a domain expert in order to solve the problem.
Conversely, there is not always a lot of sophisticated computation required to
solve the problem. There may also need to be a great deal of data provided in
order to complete the solution.

At this point, it may be wise to distinguish what we mean by knowledge
and data. By knowledge we mean information about a class of problems that
can be stated by a human expert without regard to a particular problem in
that class. Data refers to information about a particular instance in that class.
Data is, in fact, what is used to identify the particular instance of the class of
problems. Data is information which may become known to the agent solving
the problem in real time, although there are certainly examples of data that
do not become known in real time. Knowledge, however, is always available
outside of real time.

Returning to the discussion of knowledge-intensive domains, the distin-
guishing feature of such domains is the requirement for knowledge, but in
many such domains there may also be a significant need as well for data to
help in effectively applying the knowledge. The biggest advantage to using a
knowledge-intensive domain is that if one successfully applies a particular
technology to the domain, and the domain experts “sign off” on the effective-
ness of the solution, it is a strong endorsement of the validity of the solution.
Another advantage is that simply by virtue of using a real-world domain, it
may actually be easier to isolate the aspects to the solution that are domain-
independent from those that are not, because anything provided by the
expert is domain-dependent. As information is gleaned from the human
expert, that information can be represented in some form as the knowledge
1.2 Types of Domains Used In AI

appropriate to the solution. All other aspects of the solution would then fit naturally into the domain-independent part of the solution. By contrast, the biggest disadvantage to using a knowledge-intensive solution is that getting the information required from a domain expert can be very costly and frustrating. Additionally, the experts often will not agree amongst themselves about what the appropriate solution is, making it very difficult to do knowledge acquisition in a timely fashion. This disagreement may, however, produce an interesting side effect that is worth noting. The effort involved in attempting to analyze their domain to formalize it may be something the experts have never done before, and in making that effort and seeing what disagreements it produces, the experts may learn things about the domain that previously they had not thought of. Thus, it is intended that the ideas presented here have applicability not only to technology researchers interested in designing intelligent agents for real-time domains, but also to experts in real-time domains who are interested in formalizing their problems in those domains to learn more about them. A final disadvantage or difficulty encountered in knowledge-intensive domains is that it can be difficult to know how best to represent the knowledge once it is obtained.

On this particular axis, the third category of domain is compute-intensive domains. A compute-intensive domain is one that requires a great deal of computer time relative to the amount of data it processes, whereas in the case of a knowledge-intensive domain, once you have the data and the knowledge it is often a relatively trivial matter to come up with the solution (the hard part is getting and representing the knowledge). In the case of a compute-intensive domain, there is still a great deal of hard computation ahead even when you have all the necessary knowledge and data. The biggest advantage to using compute intensive domains is that they tend to be domains in which it is absolutely essential to use advanced technology to solve them, and hence the problem is not so much convincing people to use the technology once it becomes available, as it is producing it in a manner that is technically feasible. Another advantage is that they tend to be domains in which it is possible to evaluate the agent’s performance easily—it is very clear when the agent is doing well and when it is not. Conversely, there are two big disadvantages to using compute-intensive domains. One is that they often lend themselves to specific algorithms that, though very clever, are particular to the problem or problem class and cannot be applied to different domains. Since so much time is spent on computation, the biggest effort is in making that computation efficient, and that effort must generally be specifically tailored to the problem. Another disadvantage is that simply because the problem is so compute intensive, there is the real risk of very weak per-
formance if the technology is not up to solving the problem properly with the resources available to it. This is less of an issue with knowledge-intensive domains, because they are less likely to lead to a “combinatorial explosion,” and so graceful degradation of performance is more likely as the agent gets close to the limits of its ability.

The second axis on which to divide domains is the real-time versus non-real-time domains. As noted on the first page of this book, by “real time” we mean that there is some type of deadline by which the agent is required to respond. Some explanation and refinement of this definition may be in order here, though. Isn’t an agent always required to respond in real time—that is, by a deadline? Does an agent ever really have infinite time available to it in order to respond? The answer, of course, is that the agent never really has infinite time available to it. However, there are certainly occasions when the agent’s resources are sufficiently large enough that its ability to respond by the deadline is never in doubt. A better definition of “real time,” from our point of view, is: the agent is required to respond by a deadline even if the response is less desirable than what could have been achieved if the agent had had more time available.

This point bears dwelling on a little bit because it is really central to the entire book. Computers traditionally have been seen as tools that come up with a single right answer. Even if the algorithm they employ is fairly complex, and requires the best human minds in the field to state in its entirety, there ends up being a single best answer that the computer is supposed to come up with. What we are describing in this book is work geared toward designing intelligent agents that do not have just a single best answer that they are aiming at, but rather there is an entire set of possible responses that can be produced. The more resources that the intelligent agent has available to it, the better the response will be. This point is tricky, because people have become conditioned to expect perfection from computers, even when they know that they cannot expect perfection from their fellow humans. What we are suggesting is that the range of problems to which intelligent agents can be applied could be vastly increased if people became more accustomed to the idea of a graceful degradation in the computer’s performance, and were no longer insistent on perfection. If you insist on perfection, not only do you limit a tool’s application to areas in which it is the perfect tool, but you also are in an awkward position when you are unable to define perfection.

Real-time domains, therefore, are domains in which not all the inputs necessary for the agent to respond effectively are initially known. The agent has a limited amount of time to produce an output, and is generally better off coming up with a good—but necessarily imperfect—solution by a deadline than
by failing to act. The agent has some information available to it about when the deadline occurs, but cannot necessarily predict perfectly when the deadline is going to occur. The agent may be working on multiple problems at the same time, and those different problems may have different deadlines. The agent does not necessarily stop working on a problem when the deadline is reached. Rather, it does the best action available to it at that time, but continues to monitor the situation and possibly takes followup actions. The agent has to decide how to allocate resources for the solution of problems on multiple levels. It must decide how to divide computational resources between the different problems it is working on. Additionally, the solution of those problems may require resources from the “outside world,” which may have certain costs associated with them, or which are available only in limited quantity. The agent must decide how to allocate those resources. The agent must also decide, based upon changing conditions in the world, whether the allocation of computational resources should change, and it must also decide how much of its time to devote to the metaproblem and how much time to work on solving each individual problem. It must also have a way of detecting new problems when they arise so that it can allocate resources to their solution.

The primary advantage to using real-time domains as a testbed for AI techniques is that they are very rich domains where any technique will be fully tested. As will be seen later, even AI techniques which do not in and of themselves have a real-time flavor to them can often be “real-time-ized,” and then applied in a real-time setting. Another reason for using these domains is that there are probably many more real-time domains than nonreal-time domains, especially in this era of the World Wide Web and users who expect instant response, so the range of possible application of ideas is much greater if the ideas can be applied to real-time domains.

The biggest disadvantage to using real-time domains is that they can be very complex, and if the agent cannot properly get a handle on that complexity then it may flounder and be unable to do anything useful at all. Another related disadvantage is that because of this complexity, it can be difficult to properly design experiments that really test the agent’s performance. However, we believe that the time is ripe for intelligent agents to be designed that can properly cope with real-time constraints, and we hope to illustrate some techniques that may take away some of the complexity.

By contrast, nonreal-time domains are domains in which the agent is given a fixed set of inputs, virtually unlimited resources, and goes off by itself and comes back with the answer when it is done. This is something that tends to work very well in a batch mode, say as part of an overnight run, where there may be several hours during the night when the agent has to complete its
task. If it averages some amount of time considerably less than this in the completion of the task, then the time constraint presents no serious challenge to it, and it can operate essentially oblivious to real-time constraints.

The big disadvantage to non-real-time domains, though, is that they are becoming fewer and farther between, and for any technique to operate successfully it needs to be able to respond effectively in real time.

The following table summarizes the advantages and disadvantages of the different types of domains.

<table>
<thead>
<tr>
<th></th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Toy</strong></td>
<td>• Easy to design experiments</td>
<td>• “Solution in search of a problem”</td>
</tr>
<tr>
<td></td>
<td>• Compare results by different researchers</td>
<td>• Artificially constructed domains</td>
</tr>
<tr>
<td></td>
<td>• Less reliance on domain expert</td>
<td></td>
</tr>
<tr>
<td><strong>Knowledge-intensive</strong></td>
<td>• Domain expert can be a powerful endorsement</td>
<td>• Costly to get knowledge from domain experts</td>
</tr>
<tr>
<td></td>
<td>• Easier to isolate domain-independent parts of solution</td>
<td>• Disagreement amongst main experts</td>
</tr>
<tr>
<td></td>
<td>• Domain expert disagreement may lead to greater expert understanding of domain</td>
<td>• Knowledge representation problem</td>
</tr>
<tr>
<td><strong>Compute-intensive</strong></td>
<td>• Essential to use advanced technology</td>
<td>• Specific, domain-independent algorithms</td>
</tr>
<tr>
<td></td>
<td>• Easy to evaluate performance</td>
<td>• Very weak performance if the technology is insufficient</td>
</tr>
<tr>
<td><strong>Real-time</strong></td>
<td>• Very rich testbeds</td>
<td>• “Thrashing” if the complexity overwhelms the agent</td>
</tr>
<tr>
<td></td>
<td>• Wider range of applications especially in view of the Web</td>
<td>• Experiment design is difficult</td>
</tr>
<tr>
<td><strong>Nonreal-time</strong></td>
<td>• Simple to set up in batch mode</td>
<td>• Such domains becoming rare</td>
</tr>
</tbody>
</table>

As should be clear by now, the real-time/nonreal-time axis and the toy/knowledge-intensive/compute intensive axis are two separate axes into which AI domains can be divided, and it is therefore possible to further subdivide the field into six domains by taking the cross product of these two axes. This is shown in the following table, together with specific examples of such domains.
1.2 Types of Domains Used In AI

The scope of this book is primarily restricted to knowledge-intensive, real-time domains. Obviously, since the subject matter is real-time agents, it is appropriate to focus on real-time domains. And to provide the best possible testbed for ideas, the focus should be on knowledge-intensive domains.

More specifically, we have chosen to illustrate most of the ideas in this book with ideas from the aviation domain. Why choose this domain in particular, when there are other knowledge-intensive, real-time domains? Certainly the ideas presented here do apply to other domains, and there are points when we will provide references to material in other fields. However, aviation seemed like an especially good domain to use because the examples can easily be understood by a layman—it is not necessary to be an aviation expert in order to understand the examples. If we were to use medicine, for example, then it might be difficult for anyone other than a medical specialist to follow the terminology.

There has been some work by Degani (Degani & Kirlik, 1995, and Degani & Wiener, 1997) that attempts to isolate the real-time properties of the aviation domain. Degani proposes a mode-based approach to aviation. For example, the agent might be in one of several modes: “altitude capture,” “altitude hold,” or “vertical navigation” mode with regard to controlling the vertical elevation. If air traffic control has assigned an altitude of 11,000 feet to the aircraft, and the aircraft is at 8,000 feet, then it will go into “vertical navigation” mode in order to get to 11,000 feet. Once it begins to near 11,000 feet, it goes into the transition “altitude capture” mode to transition to level flight. Once it achieves level flight at 11,000 feet, it goes into “altitude hold” mode and retains that altitude. Other modes might be used for other aspects of flying, such as controlling the speed or the heading of the aircraft.

This type of work seems to represent the first step in automating the operation of an aircraft, but what Degani appears to do is provide an appropriate response once it is known what mode the pilot is in. A big part of the chal-
lenge, however, is in identifying the mode that the pilot is in. Suppose, for example, that the pilot is in the "vertical navigation" mode but sees that continuing to climb will take him into the path of another aircraft. What mode is he in then? Should he switch modes to respond to this unexpected contingency? We will attempt to answer such questions, at least partially.

It should be stressed again that this is not a book about any particular domain; the aviation domain has been chosen because of its richness and accessibility. In the aviation domain, the pilot normally constructs a flight plan prior to becoming airborne. Thus, aviation is a good illustration of AI planning techniques. We go beyond conventional planning in that we discuss conditional planning: what happens if something goes wrong with the pilot's initial plan? Should the pilot construct an alternative, conditional plan even before commencing the flight? Or should the pilot wait until a problem arises during flight and then replan to take that problem into account? Or will the pilot not have enough time available to do any serious replanning, and hence should he instead simply respond quickly and reactively to contingencies that arise during flight? Which contingencies should the pilot even care about? If the pilot spends too much time thinking about and checking for contingencies which have little chance of arising during a real flight, will he devote enough time to the actual operation of the aircraft?

Pilots normally operate not only with a plan but also with a set of checklists (Cessna, 1978). The plan may be seen as roughly corresponding to AI planning, and the checklist roughly corresponds to AI reactive systems. However, this approach is rudimentary in the sense that there is not a clear technique for the automated construction of flight plans, and checklists tend to vary somewhat depending upon circumstances—which is not made clear in the checklists themselves. The point here is that pilots already are observing some of the techniques described in this book; to automate these techniques requires that they be formalized to a degree.

1.3 Conventional Planners

We now describe in a bit more detail a number of conventional planners that have attracted attention.

1.3.1 STRIPS

Probably one of the most famous early planning systems—indeed a seminal paper in the early history of planning—is STRIPS (Fikes & Nilsson, 1971).
Like GPS, STRIPS relied upon formal theorem proving. However, it represented a significant advance on GPS in that it relied on formal theorem proving only to prove facts within a given “world model.” For search within the space of world models, heuristic search rather than theorem proving was used.

In STRIPS, Fikes and Nilsson were using the term *world model* to refer to what we are calling, in the above definition, a *state*. They represented all states as well-formed formulae (wffs) in first-order logic. For example, in an aviation domain, there might be atomic formula to state that A is an airplane, B is an airport, and so on, as well as nonatomic formula to state that if A is on the ground at airport B, then the altitude of A must equal the field elevation of B, and so on. A state or world model represents a snapshot of the world at a particular point in time, although it may include general knowledge about the world that is not subject to change over time.

Change in the world is effected through the use of operators. An “operator,” in the vernacular of Fikes and Nilsson, is similar to what we called an “action” in Definition 1. When an operator is applied to a given world model, a new-world model is produced. The properties of the operator are used to determine how the new-world model is derived from the old-world model. Thus, there is normally a precondition to the operator: a single wff that must be satisfied in the starting world model before it is legal to apply the operator at all. Second, there is a set of “delete” wffs—wffs that must be deleted from the old-world model to produce the new one. Finally there is a set of “add” wffs—wffs that must be added to the old-world model to produce the new one.

In an aviation domain, an operator might be for a given airplane to take off at a given airport. To apply this operator, a precondition would need to be met: the airplane would need to be at the airport to begin with. To get from the old-world model to the new-world model it would be necessary to delete any wffs stating that the airplane is on the ground, and add a wff stating that the airplane is airborne.

One difficulty with the STRIPS system was a certain looseness to how operators are defined, especially as regards the delete list. Deleting a wff from a world model is *not* the same thing as adding its negation to the wff. There seems to be, in each world model, an assumption that if the precondition to an operator is met, then the wffs in the delete list will actually be found in the world model. If this turns out not to be the case, though, what is to be done? For example, if the delete list asks us to delete two atomic wffs, and neither atomic wff is to be found in the world model but the conjunction of them is in the world model, should that conjunct be deleted? It seems conceptually that it should, but this type of question was never fully ironed out in STRIPS.
Once one had the notion of a world model, as well as a set of operators for manipulating world models, the actual planning process could begin. Planning required an initial world model as well as a goal. Here there is no precise correspondence to our notion of states, because inherent in this is a certain asymmetry in STRIPS. The initial world model was always given in complete detail, but the goal could be a single wff (if one only cared about a particular aspect of the goal state): it was not necessary for the goal to be a complete world model at all. In an aviation domain, therefore, the initial world model might include a complete enumeration of the location of all airplanes and all airports, while the goal would be only that a particular airplane get to a particular airport. Nevertheless, given this asymmetry, STRIPS will attempt to perform means-ends analysis. That is, it will use a theorem-prover to determine which operators can be applied in the initial world model, as well as which operators would yield the final goal state. It then attempts to determine, heuristically, which of these operators reduced the “distance” from the initial world model to the goal state, and work on those first. When it has finally found a path from the initial world model to the goal, STRIPS is done.

Thus, for example, in our aviation domain, if the airplane were at a particular airport in the initial world model, and the goal were for the airplane to be at a particular (other) airport, then first all available operators would be tested to determine whether they were applicable at all in the initial world state. Then a heuristic would be used to determine which operators appeared to be moving one closer from the initial state to the goal, and those operators would be tagged for further search.

STRIPS represented, historically, a good first step in the development of automated planners. It is, from our point of view, a purely classical planner; indeed it might even be viewed as the canonical classical planner. The outcomes of all operators are known in advance, the initial world state is known completely, and there is no allowance made for the possibility that the plan may not proceed as initially laid out because of uncertainty in the domain.

Beyond the fact that it does purely “classical” planning, STRIPS had other weaknesses. One difficulty is that it does very little actual “planning.” Generally, when humans plan out activities, they start at a certain level of abstraction, and work out a general plan. Then as the general plan begins to take shape, that plan is refined into more and more specific versions as more information becomes available. STRIPS requires one to work at the atomic level right from the beginning. A pilot using STRIPS in its original form would need to begin thinking of atomic actions, like adjusting the airspeed of his aircraft to a particular recommended setting, before the route for his
1.3 Conventional Planners

flights was even determined. Such concentration on too much detail too early severely limits the usefulness of STRIPS.

1.3.2 ABSTRIPS

Both of these significant early drawbacks to STRIPS were addressed, to a degree, in followup planners that were published in the years immediately after the original STRIPS paper. The ABSTRIPS (Sacerdoti, 1974) system allowed for planning to commence at a more abstract level, and didn’t require early concentration on detail. Triangle tables were introduced into STRIPS to allow for the monitoring and execution of a plan in progress.

ABSTRIPS cast planning as a process of planning in a series of abstraction spaces. The central observation of Sacerdoti was that planning is a search problem, and that STRIPS, by relying purely on heuristic search, is not going to be able to find a path from initial to goal state in a reasonable amount of search time for problems of any complexity. The basic idea of ABSTRIPS was fairly simple: an abstraction space of the space in which one actually wants to do planning is one in which some of the preconditions are relaxed. The idea is that one constructs a rough plan that ignores many of the required preconditions, and then with that plan you start to introduce preconditions, and gradually refine the plan into something where at each step all the preconditions in the original space are actually met.

ABSTRIPS begins the planning process by planning from the initial state to the goal state in a high abstraction level. This gives it what is known as a skeleton plan. It then attempts to determine whether this abstract skeleton plan actually works as a plan at the next lower abstraction level. Because the only difference between the abstraction levels is the relaxation of preconditions, the only way it can fail to be a valid plan at the next level down is if at some point(s) in the skeleton plan, the more specific preconditions are not met. This is resolved by additional planning at the lower abstraction level to actually meet the preconditions. Once this is done, one has an abstraction plan at the lower level, and by repeating this process indefinitely one eventually completes a plan in the real “ground” space.

By way of example, in the aviation domain previously discussed, in the ground space there might be an action $\text{FlyDirect}(a,x,y)$, meaning for airplane $a$ to fly from $x$ to $y$. In the most abstract space, the only precondition for this action might be that the airplane be at point $x$. This will allow the construction of a rudimentary flight plan. However, additional preconditions will be gradually introduced at lower level spaces. At the next level, for example, it might be required that the airplane be airborne before attempting to fly from
x to y. The earlier plan will then be refined by requiring the attainment of this precondition if the airplane is on the ground—this will introduce another action into the plan, say TakeOff(a). At lower levels still, additional preconditions would be introduced such as, perhaps, requiring air traffic control clearance, adequate fuel in the plane, etc. Once all relevant preconditions have been introduced and the planning is being done at the ground space, a complete plan will be developed.

Notice how this reduces the complexity of the search through the space. If search started in the ground space, then the number of actions required to form a complete plan would be so great that it would likely be impossible that it could be found with undirected search. Even with heuristic search of some kind being used, it would be difficult. The reason is that it is hard to see precisely what heuristic could take into account the relative values of performing different actions—such as fueling the plane, flying from one point to another, and so on—in some meaningful way that would reduce search complexity. However, by doing the search in abstraction spaces, each planning problem is one that can be solved with only a few steps, and therefore it becomes far more reasonable to believe that the complete problem can be solved. Planning is reduced to a form of island-driven search.

ABSTRIPS falls very close to the “conventional” end of the conventional-reactive continuum. However, it is not quite as pure a conventional planner as the basic STRIPS system. The reason for this is that by reducing the number of preconditions involved in the actions, they become “applicable” to a larger number of states. This gives it just a little of the flavor of a reactive planner. It is clearly far more conventional than reactive, but it represents a small step in the evolution toward reactive planners.

1.3.3 Triangle Tables

A much more reactive planner is STRIPS with triangle tables (Fikes, Hart, & Nilsson, 1972). Indeed triangle tables were probably an idea well ahead of their time in the sense that they are far more “reactive” than many other systems developed in the 1980s. The biggest difficulty with STRIPS is the perfectly deterministic assumptions that it makes about the world. It develops a plan assuming actions that are perfectly deterministic. If it can then carry out this plan in the real world, all is well; but if the behavior of the real world is even slightly different from what the STRIPS model is assuming, then it is out of luck. Triangle tables were designed to change all of this.

The idea behind a triangle table is that all operators have preconditions. If we know at a given point in the plan that not only are the preconditions for
the next operator satisfied, but also the preconditions for all subsequent operators will be satisfied at the appropriate time, then we can be confident that we are on track to complete the plan. If something unexpected happens, and the results of an action cause something to change, then we will need to revise this understanding at a later point, but the triangle table provides a good mechanism for monitoring this. More specifically, the triangle table approach starts with a set of clauses—called the initial kernel, or kernel 0—that need to be true in the initial world state for the plan to have a chance of being executed. Likewise, after the first operator in the plan is applied, a new kernel called kernel 1 will need to be true in order for the remainder of the plan to be executed. This process continues until the end of the plan: kernel \( n \) represents the set of all clauses that need to be true in order for the remainder of the plan after step \( n \) to be executed. The idea is that in the normal course of events, application of the \( n \)th operator will transform a true kernel \( n \) into a true kernel \( n+1 \). If at any point in the plan the appropriate kernel is not true, then it is necessary to replan a new course of action for getting to the final goal.

<table>
<thead>
<tr>
<th>AT(a,x)</th>
<th>OnGround(a)</th>
<th>Takeoff(a,x)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>AT(a,x)</td>
<td></td>
<td>Airborne(a)</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>AT(a,y)</td>
<td></td>
<td>Fly(a,x,y)</td>
</tr>
<tr>
<td>3</td>
<td>Airborne(a)</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>AT(a,y)</td>
<td></td>
<td>Land(a,y)</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>AT(a,y)</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>OnGround(a)</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 1.2  Example Triangle Table**

An example triangle table is shown in Figure 1.2. Here the triangle table is used for executing a three step plan in a simple aviation domain. The airplane is on the ground at point \( x \) and the pilot wants the airplane on the ground at point \( y \). The three steps are to take off at point \( x \), fly from point \( x \) to point \( y \),
and land at point \( y \). The three operators in the plan are shown above the various boxes: Takeoff\((a,x)\), Fly\((a,x,y)\), and Land\((a,y)\). The kernels in this plan are rectangular subgrids of the above triangle. Specifically, the kernels are:

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel 0</td>
<td>1,2,3,4</td>
</tr>
<tr>
<td>Kernel 1</td>
<td>2,3,4,5,6,7</td>
</tr>
<tr>
<td>Kernel 2</td>
<td>3,4,6,7,8,9</td>
</tr>
<tr>
<td>Kernel 3</td>
<td>4,7,9,10</td>
</tr>
</tbody>
</table>

The leftmost column of the triangle table represents those clauses in the initial world model that are preconditions of the various operators in the plan. It does not include clauses that are preconditions of later operators that were added by earlier operators. Thus, for example, AT\((a,x)\) is true in the initial world model and is a precondition of both of the first two operators. All the other cells in the table represent clauses added by the action at the top of the column that have not been deleted by any action up to but not including the action at the right of the row. Thus, for example, the action Takeoff\((a,x)\) causes the airplane to become airborne, so Airborne\((a)\) appears in cells 5 and 6. However, the final action, Land\((a,y)\) causes the clause Airborne\((a)\) to no longer be true, so that is why this clause no longer appears in cell 7.

As mentioned earlier, the idea behind a triangle table is that it allows the monitoring of a plan to see whether it is proceeding properly. The idea is that if all goes smoothly, the application of operators 1, 2, and 3 will in sequence transform kernel 0 into kernel 1 and then 2 and 3. So, for example, since Airborne\((a)\) is one of the clauses appearing in kernel 1, the application of operator 1 should cause this clause to be true. If for some reason at this point in the plan, this operator is not true, then replanning is necessary. This could, for example, happen if the pilot decided to abort the takeoff because of mechanical difficulties. At this point, the original plan is no longer valid, and replanning is necessary.

Triangle tables represent a step along the evolution from conventional to reactive planners—they are much more reactive in flavor than ABSTRIPS, for example—but they are still more conventional than reactive. The reason is that planning still proceeds on conventional lines, and when there is a problem with the conventional plan, then replanning can be done. The assumption is still being made that usually the effects of operators are known in advance, but the allowance is made for the fact that sometimes this will not be the case. As indicated in Figure 1.1, this is one of the distinguishing features of a reactive planner, but another distinguishing feature is that they are able to respond to a large number of states, and the plan in a triangle table basically just divides the world into two states: the “correct” one for a given
1.3 Conventional Planners

point in the plan, and everything else (which requires replanning). A more reactive planner would be able to respond to different states at a much finer level of granularity.

1.3.4 NOAH

Moving back to purely conventional planners, the next step after ABSTRIPS in the development of abstraction planners was NOAH (Sacerdoti, 1977). Like ABSTRIPS and STRIPS, NOAH is conventional, but knowledge about the domain is stored procedurally rather than declaratively in a set of what Sacerdoti calls SOUP (semantics of user’s problem) functions. SOUP functions map an incomplete plan into a more complete plan. These functions may vary depending upon the particular domain. The planning is hierarchical in the sense that NOAH starts with only an initial goal, and gradually refines the plan to produce a complete set of actions for achieving this goal. However, it differs from ABSTRIPS in a few important ways. One, the hierarchy is determined not by different levels of preconditions, but implicitly by the makeup of the SOUP functions. Second, whereas in ABSTRIPS all partial plans were stored as a totally ordered sequence of actions, in NOAH partial plans may only be partially ordered. Finally, a set of critics exists to help resolve conflicts caused by this partial ordering.

The basic algorithm of NOAH is to apply SOUP functions to determine a partially ordered set of actions to achieve the initial goal, then to apply critics to resolve possible conflicts, then apply the SOUP functions again to do further planning, and so on until a complete plan is achieved.

As before, we illustrate with an example from the aviation domain. Figure 1.3 illustrates the same example as Figure 1.2. At the beginning of the planning process, step a, the planner has to achieve two goals that are expressed as a conjunction: get the airplane to point $y$ and get it on the ground. A SOUP function will expand this conjunction into the partially ordered plan in step b. In this plan, it is known that the airplane will need to achieve both of these goals. At this point, a critic notices that the goal of getting the airplane to point $y$ will need to be achieved before the goal of getting it on the ground. This is procedural knowledge stored in the critic: the critic knows that airplanes are not mobile after they have landed, and so it orders the partially ordered plan in step c. At this point, actions to achieve the two goals are substituted for the goals themselves, step d. Here, another critic will notice that to fly, the airplane needs to be airborne, and will set a subgoal to achieve this, step e. At the end of the planning process, step f, an action to achieve this subgoal has been introduced into the plan.
Figure 1.3  NOAH Example
1.3 Conventional Planners

As is probably clear, much of the functionality of NOAH lies in procedural knowledge within the SOUP functions and critics, and these are domain-dependent, so the actual behavior could be different for a different set of functions and critics. Nevertheless, this example should give the reader an idea of the major advance provided by NOAH: hierarchical but completely conventional planning, with partial plans and a partial, ordered set of actions.

1.3.5 SIPE

The next planner to be looked at is SIPE (Wilkins, 1984). Like NOAH, SIPE is a hierarchical planner allowing parallel (partially ordered) plans. Although it has been claimed that NOAH is domain-independent, SIPE is more clearly domain-independent: NOAH sweeps much under the rug of the SOUP functions and critics. But perhaps the biggest advance of SIPE is the use of resources. The idea is that certain operators may require a particular resource. For example, all steps in the landing of an airplane may require the resource of the runway be made available. When this resource is in use, other actions requiring this resource, such as the takeoff of another airplane, would not be able to be executed.

Another advance of SIPE is the use of constraints. The notion of constraint in SIPE is actually fairly limited in the sense that a constraint is a restriction on an unbound variable in a partial plan, for example that it be a member of a particular class. Nevertheless this represents an important advance over previous domain-independent planners.

SIPE also used a notion of deductive operators. A deductive operator is an operator that allows the planner to deduce certain facts without them needing to be represented explicitly. For example, the following is a deductive operator:

DEDUCTIVE OPERATOR: POS1
ARGUMENTS: PLANE1, PLANE2, RUNWAY
TRIGGER: (ONGROUND PLANE1);
PRECONDITION: (POSITION PLANE1 RUNWAY 1),
(POSITION PLANE2 RUNWAY 2);
EFFECTS: (POSITION PLANE2 RUNWAY 1)

This operator is triggered when its trigger condition becomes true, but only if the preconditions were true just prior to the trigger condition becoming true. In this case, this means that if PLANE1 was in runway position 1, and PLANE2 was in runway position2, and then PLANE1 lands, then PLANE2 will move up to runway position 1.
SIPE also has a notion of helpful and harmful interactions on different branches of a partial plan. Specifically, SIPE uses parallelism in the development of its plans, and tries to keep as much parallelism as possible while developing a plan. However, it has to know about the interactions between different branches of a plan to correct harmful interactions, and take advantage of helpful interactions. A helpful interaction is defined as one in which a goal on one branch of a plan is made true by another branch of the plan. A harmful interaction is defined as one in which a goal on one branch of a plan is made not true by another branch of the plan.

SIPE does not appear, at least in our view, to provide a radical advance on previous planning systems; it represents more of an incremental advance in the planning literature. Nevertheless, it is probably one of the strongest classical planners in the sense that formalized things such as helpful and harmful interactions, constraint handling, and management of resources had previously been dealt with in only an ad hoc fashion. On the reactive-conventional spectrum, however, SIPE is clearly at the conventional end.

Other domain-independent planning systems, mentioned for the sake of completeness, include NONLIN (Tate, 1977), DEVISER (Vere, 1981). NONLIN’s contribution was to introduce a form of backtracking, and DEVISER took the backtracking process a step further.

1.3.6 TWEAK

The last conventional planner that we look at in some detail is TWEAK (Chapman, 1987). The reason this planner bears special note is that it was an attempt to impose a complete formal logic on the conventional planning process. As such, it both represents a substantial advance on previous conventional planners, and also illustrates the inherent limitations of conventional planners.

Chapman originally was not interested in planning—his interest was machine learning—and he just wanted to include a planner within his learning system to make an integrated problem solver. However, he found that it just wasn’t feasible to insert NOAH into his system as a black-box subroutine—it was just too ad hoc. Chapman therefore designed his own planner to be a completely formal solution to the nonlinear planning problem.

Chapman sees planning as the process of making one or more propositions true in a given situation. Much of the paper on TWEAK is centered on the “Modal Truth Criterion,” which states the general conditions under which a proposition is true in a given situation. Essentially, a proposition will be true in a given situation if it must be asserted sometime not after the current situ-
1.3 Conventional Planners

An example from the aviation domain is again appropriate. One step in a plan might be to land the aircraft, and this step would require that the proposition (clear runway 22) be true. This proposition might have originally been asserted by the action of another plane taking off from runway 22. After the other plane took off, this runway would be clear. This means that the pilot is free to land, except in the case where some action occurs such as a third plane pulling out onto the runway, which would cause the proposition (clear runway 22) to not be true. As stated before, such an action is called a clobberer, and it prevents the pilot from carrying out his action. All is not lost, however, if it can also be guaranteed that there exists a white knight—the third plane taking off—which reasserts the desired proposition. The “modal truth criterion” basically states that as long as you can guarantee the existence of a white knight occurring after every clobberer, but before the situation in which the proposition is required to be true, then the proposition will be true.

In TWEAK, then, the planning process is seen as repeated attempts to satisfy the modal truth criterion. Chapman presents a nondeterministic algorithm for satisfying the modal truth criterion. Every planning problem is originally represented as the need to satisfy a set of propositions. Gradually—through repeated efforts to satisfy the modal truth criterion—additional steps are introduced into the plan. These are either original steps designed to achieve propositions, or white knights designed to reverse the effects of clobberers. Where feasible, constraints are introduced into the plan—both temporal constraints requiring one step to precede another in the plan, and codesignation constraints that specify in more detail what happens in each step in the plan.

As new steps are introduced, they require certain preconditions to be met, and that in turn requires further invocation of the modal truth criterion. Because the algorithm is nondeterministic, to actually implement it one would need to explore all possible paths through the algorithm, with a backtracking component whenever one path led to a dead end.

Chapman proves some interesting results about his planner. One is that if TWEAK does in fact terminate, then the plan it produces does in fact solve the problem. Because everything in the algorithm is based on the modal truth criterion, this basically amounts to saying that the modal truth criterion is correct. Alternatively, if TWEAK either halts signalling failure or does not halt, then there is no solution. Chapman also proves that planning is undecidable because any Turing machine can be encoded as a planning problem in
TWEAK. A Turing machine, as the reader will recall, is a simple abstract machine that is nevertheless capable of computing any computable function (Turing, 1936).

TWEAK is in one sense something of a strawman, because it does not appear that Chapman ever intended it to be used for actual planning problems. Nevertheless because it represents the first attempt to fully formalize the planning process, it helps to indicate the limitations of planning. For example, the fact that planning is undecidable appears to put a limit on conventional planning. Clearly it is not sufficient in the real world to simply say that a planning problem is undecidable and fail to act when action is required. Nor would it be sufficient to go into an infinite loop and not act because no action had been recommended.

The modal truth criterion also represents an important step backward from some earlier planners in the sense that it involves no notion of abstraction at all. For example, consider again the aviation domain. The modal truth criterion requires that for every potential clobberer, a definitive white knight be exhibited that reasserts the desired proposition. If the proposition again is that runway 22 be clear, then as soon as the potential exists for another airplane to use the runway, some definitive resolution of the clobberer situation is required. This resolution would have to take the form of either constraining the other plane to take off before our plane takes the runway, or alternatively requiring that we take off before the other plane takes the runway. If there were some notion of abstraction planning, this problem would not exist. The whole process of an aircraft taking the runway and then taking off would be represented as an abstract step, which would not clobber the runway being clear and would not require that the clobbering situation be resolved. However, it appears that doing away with abstraction planning is part of the sacrifice that Chapman had to make in order to guarantee correctness and completeness of his system.

1.3.7 Conclusion

The purpose of this discussion of conventional planners has been to lead the reader through to the state of the art in the mid-1980s so that the reader can understand some of the history of planning, and also the motivation for the development of reactive planners which began around the mid-1980s and represent the main focus of this book. In Figure 1.4, we show again, very roughly, how these pre-1985 planners fit on the conventional-reactive planning continuum.
1.3 Conventional Planners

The most conventional of the pre-1985 planners is probably TWEAK. TWEAK is a very formal system that provides little opportunity for reaction of any kind. Indeed, it is not even clear how one would go about revising TWEAK into a more reactive planner. STRIPS is almost as “conventional,” but it is slightly more heuristic, and therefore provided the seeds for building more reactive planners. ABSTRIPS and NOAH, by providing high level abstract operators that applied to a large number of states, are probably the next most “reactive” systems, followed by SIPE and DEVISER. All of these systems, though, are much closer to the reactive end of the spectrum. The only pre-1985 planners that were really reactive were triangle tables, because they provided a strong component for monitoring plan execution.

Planning has been important to AI because to solve any problem effectively, it is necessary to set goals and to achieve them, or to learn quickly that they cannot be achieved. Planning may be defined as the process of achieving a goal. However, conventional planners by themselves were not up to the task of achieving goals in the real world, because they were unable to respond to the type of real world contingency that always arises in practice. To build planners that are effective in the real world, much more robustness in the face of dynamic environments is needed. The remainder of this book will describe a number of such planners, which we believe have the required robustness, and also our call to action to implement these ideas in deployed systems.
We have looked at conventional planners and other planners that, while not purely conventional in nature, are fairly close to conventional planners on the reactive-conventional spectrum. There are two other large groups of planners that bear looking at in some detail. These are systems that do pure (hard-wired) reaction, and systems that can be considered conventional real-time planners. Both types of systems are capable of responding to contingencies that arise in real-time. The difference between the two types of system is that whereas real-time planners are also capable of doing some planning at real time, systems with hard-wired reaction are capable of doing only reaction at “run time,” and any planning that they do is done at “compile time.”

1.4.1 Subsumption Architecture

Brooks’ subsumption architecture falls into this general category of hard-wired reaction (Brooks, 1986). The idea behind the subsumption architecture is that the behavior of an intelligent agent can be subdivided into a number of levels of competence. The lowest level of competence will provide some very basic behavior, and each higher level of competence provides a higher level type of behavior. The way this is accomplished is that all levels of behavior provide reactive response to various inputs from the world. However, the higher levels of competence are also able to inject data into the inputs of the lower levels to alter their behavior; that is how the different levels of competence interact with each other. The lower levels of competence, however, do not attempt to alter the behavior of the higher levels of competence and do not even know about the existence of those levels of competence.

Following is an example of what the different levels of competence could be in an aviation domain.

<table>
<thead>
<tr>
<th>Level</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Keep aircraft in controlled flight</td>
</tr>
<tr>
<td>1</td>
<td>Avoid other aircraft</td>
</tr>
<tr>
<td>2</td>
<td>Execute a flight plan</td>
</tr>
<tr>
<td>3</td>
<td>Communicate with control tower</td>
</tr>
<tr>
<td>4</td>
<td>Execute control tower instructions</td>
</tr>
<tr>
<td>5</td>
<td>Replan, based on contingencies</td>
</tr>
</tbody>
</table>
Thus, the first level of the architecture tells the agent simply to keep the aircraft flying—that is, to avoid stalling the aircraft. The next level up deals with the task of avoiding other aircraft. Because there are many possible ways to execute level 0, keeping the aircraft in controlled flight—this behavior is underconstrained and can be additionally constrained by the outputs from the second level, which basically decides that the aircraft must remain in controlled flight in such a way that it also avoids other aircraft. Additional levels of competence, as one moves further up, provide steadily more sophisticated behaviors.

The subsumption architecture was designed to meet a variety of requirements. These requirements included the need for multiple goals. Thus, in the above scenario, the pilot may have a goal to avoid stalling the aircraft at the same time as he has a goal to fly to a particular destination airport. The idea of the subsumption architecture was in part to allow the interplay between these different goals to happen as seamlessly as possible. In this case, the higher level (flying to the destination airport) would probably be implemented by altering the input to one of the lower levels, so that the lower level goal would take precedence at execution time, which is the type of behavior one would want. The robot was also expected to be robust. If some sensors fail, it should be possible for the robot's behavior to degrade gracefully rather than fall apart completely. It is less clear how this graceful degradation is achieved in the subsumption architecture.

Brooks provided his own example of the use of a subsumption architecture, where the first three levels—avoid objects, wander, and explore—have some similarity to the first three levels in the aviation example given above. Brooks indicates how the specifics of the subsumption architecture would be constructed for these first few layers. It is not altogether clear, however, how the approach scales up as additional layers are added. There is a very strong hard-wired flavor to what Brooks is doing, and the precise inputs in lower level layers that the higher levels alter needs to be hard wired. It is not clear that one doesn't get a combinatorial explosion of connections between the different layers as the more complex higher layers are added, which is something that Brooks is definitely trying to avoid (and claiming to be able to avoid).

Within the subsumption architecture, each layer is implemented as a set of modules; each module in turn is a finite state machine. There are four categories of states associated with these finite state machines. Output states are ones in which the finite state machine produces an output on one of its output lines. Side effect states are ones where one of the instance variables (which augment the finite state machine) is altered based upon a combination of the input data and the existing instance variables. Conditional dispatch states are ones where a predicate based on the instance variables and
input buffers is computed, and as a result one of two new states is entered. Finally, event dispatch states are ones where a set of conditions, and the states to branch to, are monitored until one of the conditions becomes true and the branch is taken.

Brooks also provides a language for specifying the behavior of each individual module. The language is a dialect of the well-known artificial intelligence language Lisp (Steele, 1990), although there would be no reason it could not be written in other languages. What isn’t clear, however, is how all this scales up. Each individual finite state machine/module appears to correspond roughly to the level of an individual neuron in the human brain. If one had to individually program the behavior of each individual neuron in the human brain, the process would take far too long. But that seems to be what Brooks asks one to do. If there were some form of automated programming mechanism that augmented Brooks approach, then it would be much more believable that it could actually scale up to more intelligent behavior. In later work, Brooks does describe the possibility of using genetic algorithms to automate the programming of the behavior of autonomous robots (Brooks, 1992). It would be interesting, however, to see more examples of how this might be done than Brooks provides.

Brooks’ approach is hard-wired in a couple of different senses: First, in the sense that we usually use the term in this book—the reactions to particular contingencies are known and programmed in advance. However, it is hard-wired in another sense in that Brooks seems to be suggesting programming the actual circuitry to do particular things, as opposed to a standard computer program where one does not know in advance what address in memory the program will reside at when it is finally run (Brooks, 1997).

One conceivably might question the decision to categorize this as a hard-wired reaction type of system when the higher levels in the subsumption architecture do provide a degree of planning capability. The reason for this decision is that, so far as we know, Brooks has only implemented the first few levels of competence, and hence what he is actually able to show is hard-wired reaction. If he could actually build the higher levels, then this might be properly recategorized as a true real-time planning system.

### 1.4.2 Pengi

Another example of a purely hard-wired reaction system is Pengi (Agre & Chapman, 1987). The basic idea underlying Pengi seems to be that planning is not really required, although designing a reactive system that is capable of behaving like it is able to plan may be necessary. In many types of domains,
one has only a very small amount of time to react; the example that Agre and Chapman cite is that if you step on a rock and it pivots unexpectedly, you have only milliseconds to react and therefore have no time to prove theorems. Agre and Chapman seem to reject, for this type of domain, what they call capital-P “Planning.” From their point of view, “Planning” is something that was done in earlier AI systems where there was a smart “Planning” phase followed by a dumb “Execution” phase. People, however, engage in what they call lower-case-p “planning,” where they may have a plan to follow, but there is always a layer of improvisation going on where people are deciding what to do now based upon the state of the world now.

It seems, based upon the introduction given by Agre and Chapman, that they offer an approach to lower-case-p planning, but it is hard to see how they offer anything more than a pure hard-wired system. Their domain of application is the design of an agent that will play the video game Pengo. They basically seem to be designing what amounts to a simple rule-based system to play this video game. Their rules are generic in the sense that they refer to general situations—such as the position of the main character in the game relative to its opponents—as opposed to specific rules about how to proceed when at a particular position on the game board. The need for this type of functionality seems both necessary and obvious—if you didn’t have some type of generic rules, you would very quickly get into a combinatorial explosion of rules.

Pengi also provides a notion of action arbitration. Action arbitration refers to the fact that multiple rules may be triggered at the same time, and there needs to be some kind of control feature to determine which particular rule to fire. So Pengi provides the possibility of control rules that can determine which of a set of rules to fire in different situations. There can be higher and lower level control rules as well. Presumably Pengi provides, as well, some notion of salience so that the higher level control rules do in fact fire before the lower level rules, although this is not at all made clear in the paper.

What Agre and Chapman assert is that their notion of action arbitration provides some notion of Planning in the upper-case-p, traditional AI sense. Here we would disagree with them a bit. To the extent that it provides any notion of planning at all, it seems to be lower-case-p planning, because the planning process is being interleaved with execution of plans. Indeed, as an actual AI system, Pengi seems to be a simple rule-based system; it is the distinction between planning and Planning made at the beginning of the paper that is most interesting. What we are really proposing throughout this book is the development of a theory of lower-case-p planning, where the planning process is interleaved as appropriate with a more basic reactive system. If
Pengi’s action arbitration mechanism were fleshed out into a full control architecture, then it might provide a strong foundation for such a system. Indeed, it is quite possible that this is something that Agre and Chapman in fact do elsewhere, but it is not really made clear in their paper.

It is not clear whether Agre and Chapman provide anything beyond that which Brooks provides. They claim that their system can exhibit planful behavior, but they are operating in a domain where very little planning is involved. Also, it is not clear how they propose to scale up their approach to a planning system, whereas with Brooks it was at least clear how he proposed to do this, even if it was not clear if it would work. The main difference between the subsumption architecture and Pengi seems to be that in the subsumption architecture behavior is hard-wired in the hardware, whereas in Pengi it might be hard-wired but it is at least represented symbolically in software.

1.4.3 Action Networks

The next purely reactive system to look at is the action networks of Nilsson (Nilsson, 1988). The idea behind an action network is to encode a universal plan as a network of logical gates, called action units, that successfully implement the universal plan in a reactive manner. An example of an action unit is illustrated in Figure 1.5. There are a number of inputs to this action unit, and one output, the action A. The first input, P, at the top, refers to the “purpose” of the action unit. Each action unit has a purpose that it is trying to achieve, and the input P is used to determine whether the purpose has been achieved or not. The input at the bottom, G, refers to the “goal” input.

It is probably worth explaining what the difference is between the “goal” G and the “purpose” P. The goal G will be set to “on” when a current goal of the agent is to achieve whatever it is that this action unit is supposed to achieve.
1.4 Pure (Hard-Wired) Reaction

The purpose $P$ will be set to “on” only when this goal has actually been achieved.

In addition, the action unit has a couple of other inputs that are additional preconditions which have to be true in order for this particular action unit to be active. Thus, the action unit is an AND gate that fires when the purpose condition $P$ is false and all other inputs are true. This is illustrated in Figure 1.6.

![Diagram of an AND gate with inputs $\sim P$, B1, B2, and G, and output action A.]

Figure 1.6 An action unit as an AND gate

Once the basic building block of the action unit was defined, Nilsson went on to construct action networks. An action network is, as the name implies, a network of action units designed to solve a particular problem. In addition to the action units, the agent had at all times a set of beliefs and goals; the beliefs and goals acted as inputs to the action units. The agent also had actions that it could perform. The actions would alter the state of the world, which in turn would have an effect on the beliefs that the agent held. This connection was never made explicit, however: there was no world model expressing how the beliefs would be likely to alter the state of the world.

An example of an action network is illustrated in Figure 1.7. This action network implements the last two steps in the process of landing a plane: flying forward when on final approach aligned with the runway centerline; and then actually landing. On the left side of the action network are shown the beliefs of the agent, and on the right side are its goals. Note that “Landed” is both a belief and a goal. It is the (only) goal of the agent in this case, and when as a belief it becomes true, the agent’s work will be done.
Figure 1.7  A Simple Action Network

Action networks were a good example, at the time, of a pure reactive system. Nilsson went on to define a language called ASTRAL for defining action networks. The biggest weakness of action networks were that the action units were very primitive AND gates, and no mechanism was provided for a richer formalism. Also, the absence of any type of world model could make it tough to follow what an action network was doing, even if its actions were actually correct. These concerns were largely addressed in a later formalism that Nilsson designed, called teleo-reactive programs, described in the next section.

1.4.3 The Universal Planning Debate

An interesting debate was waged in the Winter 1989 issue of *AI Magazine* over the topic of universal plans. A universal plan is, roughly speaking, one that provides an action to be performed in all possible situations. The term was coined by Marcel Schoppers (Schoppers, 1987), but more generally has been applied to all reactive plans—such as the other reactive planning approaches described in this section. The idea of universal planning was critiqued in that issue of *AI Magazine* by Ginsberg (Ginsberg, 1989). Ginsberg defines a universal plan as a mapping from the set of possible situations into the set of primitive actions. In making this definition, Ginsberg is making a couple of assumptions of how much computation a universal plan is allowed to do. He is assuming that the universal plan must invest no computational effort in determining the action to be performed (hence the requirement that the action to be performed be “primitive”). However, he is allowing some amount
1.4 Pure (Hard-Wired) Reaction

of computation in determining the “situation” that the agent is in. That is, he does not require that all individual situations be listed explicitly.

Instead, Ginsberg assumes that universal plans are implemented with a network of gates. He also assumes that the situations in a universal plan are encoded with a set of $n$ (binary) sensors, so that there are $2^n$ situations in the plan. He then goes on to provide a counting argument intended to convince the reader that to implement such a universal plan will require on the order of $2^n$ gates, which according to Ginsberg will be infeasible except for very small values of $n$.

Ginsberg acknowledges one possible weakness in his argument. He does not actually prove that there must be $2^n$ gates to implement such a universal plan. What he states is that the average universal plan will require $2^n$ gates to implement, and that therefore the defenders of the use of universal plans need to provide some convincing arguments that the typical universal plan that one would want to implement can be implemented using far fewer gates than would be expected to be needed on average. Ginsberg then places the burden of proof on the defenders of universal plans to provide a convincing argument that this must be true, rather than on himself to provide a convincing argument that this must be false.

If we look at what Ginsberg is really asserting here, we see that he is requiring that time complexity of the universal plan be kept to a constant (hence his requirement that the plan be implemented just with a set of gates). And he wants to keep the space requirement to a minimum given the restriction on the time complexity. It seems that there might be alternate means of implementing a universal plan. For example, if the requirement that the time complexity be kept to a constant could be relaxed so that time complexity only needed to be on average small, then there would be a potentially larger range of universal plans that could be implemented. Also, if the requirement that the time complexity on the “right hand side” of the universal plan be kept essentially zero could be relaxed, then again a larger range of universal plans could be implemented. Even allowing for Ginsberg’s restrictions, it seems counterintuitive that the space complexity of the type of universal plan one would actually want to implement would be as high on average as a randomly selected mapping from the set of situations to the set of actions. To the extent that the universal plan one wants to implement has some kind of structure to it, it seems that structure could be taken advantage of to reduce the number of gates required.

Still, there are going to be problems where the amount of effort required to implement the universal plan in advance is going to be prohibitive. Hence, some replanning is going to be necessary at execution time.
Ginsberg’s critique of universal plans is itself critiqued in the same issue of *AI Magazine*. These critiques are provided by Chapman (Chapman, 1989) and Schoppers (Schoppers, 1989). Chapman’s argument is essentially that the universal plan that one would wish to construct in practice is not random, and hence has some structure to it. Ginsberg’s argument, taken to its extreme, could be used to suggest that any computer program that solves a problem is a form of universal plan within a particular domain, and therefore requires exponential spatial complexity. Since there are obviously many examples of computer programs that execute successfully in polynomial time, this would make Ginsberg’s arguments specious.

Schoppers provides a group of nine arguments in critique of Ginsberg. The first group of arguments is very similar to that which Chapman offers. Schoppers also argues that since humans seem capable of reacting effectively to situations that arise, we have an existence proof of a nonexponential reaction plan. He also argues that Ginsberg is assuming that the reaction plan has to be explicit, whereas in fact it does not.

These arguments seem strong. Schoppers also offers a weaker set of arguments based upon the idea that a reaction plan is a form of cache, and that caching is obviously an effective way to handle certain problems. This seems to be a weaker argument, because caching is effective only in certain types of problems where exact situations are likely to repeat. To make this argument in a more general way, Schoppers would need to extend it to talk about reaction plans as case-based reasoning, which he does not seem to do.

Ginsberg published a final rebuttal in the same issue of *AI Magazine*, where he boils his argument down to two points: that the behavior of an agent cannot in practice be reduced to a pure reactive plan—because there are too many behaviors one would wish to capture—and that reactive systems are unable to improve their performance by expending additional computational resources. However valid these counterarguments may be in certain cases, they seem to fall away if the use of reactive plans represent only part of an intelligent agent’s arsenal. In such a case, there is no longer any claim that the agent behavior can be reduced to a reactive plan alone, and intelligent behavior is exhibited by the agent (behavior that distinguishes agent from intelligent agent) both in the automated design of its reactive plan and in its use of other intelligent approaches. Also, we will describe an approach to reactive planning later in this book that does allow the agent to improve its behavior through expending additional computational resources.
1.5 Classical Real-time Planners

1.5.1 Teleo-reactive Programs

An example of a classical real-time planner is Nilsson’s *teleo-reactive programs* (Nilsson, 1994). A teleo-reactive program is something like a production system in that it consists of a set of productions such that a production is fired (its right-hand side [RHS] is executed) whenever its left-hand side [LHS] happens to be true. However, unlike a traditional production system, a teleo-reactive program is one designed to achieve a particular goal, and as such the productions are subject to a total order (or, in the case of a refinement of the idea known as teleo-reactive trees, they are subject only to a partial order). The highest level production is the one whose action will result on the goal being satisfied. At any given time, the production that fires is the one whose LHS condition is true and which is highest on the list.

More specifically, a teleo-reactive *sequence* would take the following form:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K_1$</td>
<td>$a_1$</td>
</tr>
<tr>
<td>$K_2$</td>
<td>$a_2$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$K_n$</td>
<td>$a_n$</td>
</tr>
</tbody>
</table>

Each condition $K_i$ must be satisfied in order for the corresponding action $a_i$ to be performed. If multiple conditions $K_i$ are true at the same time, then the production with the lowest value of $i$ is fired and the corresponding $a_i$ is performed. The action $a_i$ is the one that results in the ultimate goal becoming true.

As before, let us illustrate these ideas with an example drawn from the aviation domain. Suppose that an airplane is in the traffic pattern getting ready to land. The pilot’s goal is to achieve the condition of the airplane being on the ground. The following T-R sequence will result in this goal being achieved:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Over numbers</td>
<td>Land</td>
</tr>
<tr>
<td>Facing runway centerline</td>
<td>Fly forward</td>
</tr>
</tbody>
</table>
Note that the actions within a teleo-reactive sequence are *durative* rather than *discrete*. A *durative* action is one that continues for as long as the condition that caused it to be executed continues to be true. By contrast, a *discrete* action is one that ends after a particular period of time. Durative actions have more of a real-time flavor to them, since there is an assumption that something is monitoring the real-time behavior of the system to determine when the invoking condition ceases to be true.

There are several different concepts that Nilsson defines. The notion of the teleo-reactive sequence has already been described. There is also something known as a *teleo-reactive program* and a *teleo-reactive tree*. A teleo-reactive program is a generalization of the notion of a teleo-reactive sequence to permit the rules to contain free variables, on both the left side and the right side, which are bound when the program is invoked. The concept can be further generalized to allow the free variable to change at execution time. Continuing with the aviation example, if we add the free variable LeadAircraft to the above example, then the following would be a teleo-reactive program:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>On left base and near extended runway centerline</td>
<td>Turn left</td>
</tr>
<tr>
<td>On left base</td>
<td>Fly forward</td>
</tr>
<tr>
<td>On left downwind and at 45 degree angle to numbers</td>
<td>Turn left</td>
</tr>
<tr>
<td>On left downwind</td>
<td>Fly forward</td>
</tr>
<tr>
<td>On right base and near extended runway centerline</td>
<td>Turn right</td>
</tr>
<tr>
<td>On right base</td>
<td>Fly forward</td>
</tr>
<tr>
<td>On right downwind and at 45 degree angle to numbers</td>
<td>Turn right</td>
</tr>
<tr>
<td>On right downwind</td>
<td>Fly forward</td>
</tr>
</tbody>
</table>

Note that the actions within a teleo-reactive sequence are *durative* rather than *discrete*. A *durative* action is one that continues for as long as the condition that caused it to be executed continues to be true. By contrast, a *discrete* action is one that ends after a particular period of time. Durative actions have more of a real-time flavor to them, since there is an assumption that something is monitoring the real-time behavior of the system to determine when the invoking condition ceases to be true.

The action problem involves a single discrete state and single discrete action.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Over numbers</td>
<td>Land</td>
</tr>
<tr>
<td>Facing runway centerline</td>
<td>Fly forward</td>
</tr>
<tr>
<td>On left base and near extended runway centerline</td>
<td>Turn left</td>
</tr>
<tr>
<td>On left base</td>
<td>Fly forward</td>
</tr>
</tbody>
</table>
1.5 Classical Real-time Planners

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>On left downwind and at 45 degree or greater angle to numbers and abeamLeadAircraft or pastLeadAircraft</td>
<td>Turn left</td>
</tr>
<tr>
<td>On left downwind</td>
<td>Fly forward</td>
</tr>
<tr>
<td>On right base and near extended runway centerline</td>
<td>Turn right</td>
</tr>
<tr>
<td>On right base</td>
<td>Fly forward</td>
</tr>
<tr>
<td>On right downwind and at 45 degree or greater angle to numbers and abeamLeadAircraft or pastLeadAircraft</td>
<td>Turn right</td>
</tr>
<tr>
<td>On right downwind</td>
<td>Fly forward</td>
</tr>
</tbody>
</table>

Here, LeadAircraft refers to the aircraft that the pilot has been instructed to follow in approaching the field. The pilot ordinarily will not make his base turn until abeam that aircraft. The air traffic controller then has the potential of changing the behavior of the agent merely by altering the value of this variable. One could see that by having more free variables, much more complex agent behavior could be obtained.

The final step that Nilsson takes in this analysis is the idea of teleo-reactive trees. A teleo-reactive tree is a more general notion of a teleo-reactive program in which the order on the nodes is no longer total but now only partial. The teleo-reactive tree, when executed, executes the action associated with the shallowest node in the tree whose condition happens to be true. The above teleo-reactive program can be generalized into a teleo-reactive tree as shown in Figure 1.8. The biggest question that arises with regard to teleo-reactive trees is what happens when more than one node, neither one of which is a direct ancestor of another, has its condition satisfied at any given time. The answer, not directly addressed by Nilsson, appears to be that the behavior of the teleo-reactive tree is nondeterministic in such a case. Either (or one of all valid) actions may execute in such a case. In the case of the example in Figure 1.8, it is not possible within this domain for competing nodes on different branches of the tree to be true at any one time. There are only two branches in this particular teleo-reactive tree, and the nodes on those two branches are mutually exclusive.
The whole teleo-reactive paradigm discussed here might easily have fit in the previous section on purely reactive systems were it not for some comments that Nilsson makes near the end of the paper. Nilsson's idea is that teleo-reactive trees be automatically constructed using some form of backward-chaining algorithm from a goal node. Therefore, it is conceivable that once an agent arrives at some specific goal, it could construct a teleo-reactive tree by a backward-chaining process in real time. Coupled with such an algo-
1.5 Classical Real-time Planners

Algorithm, the whole teleo-reactive paradigm would become a genuine real-time approach, as opposed to a pure reactive methodology. However, Nilsson leaves it as a topic for future research to design such an algorithm. Such an algorithm would be similar to existing backward-chaining algorithms, but the durative nature of the actions involved in teleo-reactive trees might introduce another layer of complexity.

By and large, teleo-reactive programs subsume the notion of action networks that were described in the previous section, but there is one important way in which action networks seem more general. Teleo-reactive programs ultimately aim at making a single goal true, whereas action networks can have multiple goals at any one time, and it is not necessary in that formalism that one goal be a subgoal of another. It would be interesting to extend the teleo-reactive paradigm so that it would be able to deal with multiple goals in an interesting way. Of course, there could simply be separate programs for each goal, but if the different main goals share subgoals in common, then this would not be the most optimal program.

1.5.2 Situated Control Rules

Another example of a real-time planning system is the situated control rules of Drummond (Drummond, 1989). As an intermediate step toward the situated control rules, Drummond defines something known as a plan net. A plan net is a bipartite graph joining conditions and operators. Each condition is a predicate that is true or false in the world at any time; each operator is something that can be performed (an event). If a directed edge connects a condition to an operator, then it indicates that that condition will lead to that operator being performed; if a directed edge connects an operator to a condition, then it indicates that the performance of that operator may lead to that condition being true.

A plan net might be seen as something a bit akin to a universal plan, except that rather than particularly recommending an action in any given case, a plan net simply enumerates a number of possible actions. These actions might or might not lead to the plan’s goal being achieved. The idea that Drummond has to ensure that the goal is achieved is to additionally define something called a situated control rule. SCRs are used only under conditions where the goal is only possibly true. By “possibly true,” it is meant that some set of actions from that point will lead to the goal being true, but some other set of actions will lead to it not being true. These points Drummond calls critical choice points. It is more important to be able to make a wise decision at such points. At points where the goal is necessarily true, anything
will work, and at points where the goal is necessarily false, nothing will work. Therefore, “simply” by providing rules to govern action at these critical choice points, a favorable outcome to the plan can apparently be achieved.

An example of a plan net is given in Figure 1.9. This figure gives the same example as given in Figure 1.8, but for plan nets. As can be seen, the “Fly Forward” operator (to cite but a single example) is applied in a number of different situations, and the next step after applying the “Fly Forward” operator is therefore not clear. This is where situated control rules apply. More specifically, a situated control rule consists of an antecedent predicate and a set of sets of operators. If the antecedent predicate holds true in any given

Figure 1.9 A Plan Net (first step to Situated Control Rules)
situation, then any one of the set of sets of operators may be applied and will
lead to the successful completion of the goal. Figure 1.10 gives an example
situation control rule for the aviation example given in Figure 1.9.

![Antecedent Condition](On Left Base)

![Operator Sets](Fly Forward, Turn Left, Fly Forward, Land)

**Figure 1.10** A Situated Control Rule

The basic idea underlying situated control rules seems like a good one: construct a general plan, then find the particular places in the plan where there might be problems in successfully reaching the goal, and augment the plan with some more detailed instruction to avoid any problems that might otherwise arise. Unfortunately, it seems that with the particular formalism that Drummond offers, virtually every point in a real plan is going to be a “critical choice point” and therefore will require a detailed situated control rule. Also, under Drummond’s formalism, the situated control rule is going to be required to guarantee that the goal is met. The formalism might actually work very well if it were modified in a couple of ways. One, there needs to be a less general way of defining what the “critical choice points” are so that one doesn’t have to put situated control rules at all such points. Second, the situated control rules should simply help the agent to keep on track, not necessarily guarantee the achievement of the goal. Drummond illustrates his points with a simple example drawn from the blocks world, but even in the fairly simple aviation example we give it seems Drummond is largely out of his element. Situated control rules are a good idea that were just not developed enough.

### 1.5.3 Phoenix

Both situated control rules and teleo-reactive programs have a formal flavor to them: the formalism underlying the idea is very precisely defined. Another
system, which is more architecturally oriented, is the Phoenix system (Cohen et al., 1989). Phoenix was an agent architecture designed to deal with environments that were dynamic, on-going, real-time, unpredictable, varied, operated on multiple scales both with respect to time and space, and which were spatially distributed. It accomplished this through resource management, uncertainty management, cooperation, and planning. Phoenix divided the real-time aspects of the architecture into two large components known as the reflexive and the cognitive components. The cognitive component had a case-based reasoning flavor to it in that it used a set of stored plans, and tried to retrieve the one most relevant to a given situation and instantiate it for a given situation. The “instantiation” of a plan for a given situation was actually a fairly complex process. A plan might have an action in it that is to be performed at a certain point. There might be a number of different ways of actually carrying out this action. For example, if the action involves moving from point A to point B, then there may be different path planning algorithms that can be invoked in order to do this. The Phoenix agent knew about the time complexity of the different algorithms available and would select one based upon how much time happened to be available at any given point in the plan. There also seemed to be a strong recursive nature to the planning process in Phoenix. Many of the high level actions within plans were actions to do more planning. This allowed the possibility of interleaving the planning process with the execution of real-world actions in the plans.

In order to best choose which method to use to execute a given action, the agent would delay making that choice until the action actually had to be executed. The agent kept a timeline, which was partially ordered, of actions to perform. It would always execute the action that was next on the timeline. However, there were a couple of reasons why more than one action might be “next”: the skeletal plans that Phoenix had in its library might not completely order all actions within each plan. Also, if the agent was simultaneously executing multiple plans, then there would be no a priori order on the actions in those different plans. It was not entirely clear how the agent went about deciding which action to perform in such cases, although presumably it depended on factors similar to those used to decide which method to use to execute a plan: the real-time constraints on the different actions.

There were several different types of actions that the agent might perform. There were selection actions that resulted in a search of the plan library. Plan actions were actions that were placeholders for a plan. When a plan action was executed, it was expanded into its component parts. Those component parts often included many primitive actions that actually did computations or sent commands to sensors or effectors. A plan, however,
could contain all three types of actions. It would be entirely possible for part of a plan to be involved in solving a difficult subproblem for which the plan library would again be consulted.

The type of planning done by Phoenix was essentially a form of lazy skeletal refinement in that some decisions are not made until execution time, or until a decision is required. Indeed, a great deal seems to depend on having good skeletal plans available. If the skeletal plans are effective, and the appropriate one is selected for a given circumstance, then the performance of the agent is likely to be adequate. However, Phoenix did not seem to provide a mechanism for actually constructing the skeletal plans, and the planning mechanism seemed otherwise rather weak.

In addition to the cognitive component, Phoenix had a reflective component for dealing with unexpected situations that arise. There were three separate ways in which Phoenix could respond to unexpected events. The first, operating on the shortest time scale, were reflexes. A reflex can halt a potentially injurious action. A reflex does little processing and returns little information. A reflex does not interrupt the cognitive component, which will continue to work on whatever it is working on until it next checks the status of the agent; nevertheless the reflex is able to keep the agent functioning until this occurs.

The second approach was error recovery and replanning. An error was an unexpected event preventing the completion of an action or plan. For example, if the agent attempts to find a skeletal plan matching the required constraints but is unsuccessful, then an error will be raised and the agent will attempt to replan. The mechanism for dealing with this is to place on the timeline a “deal with error” action that, when executed, will generate a plan for handling the error. Of course, one could imagine this raising a “meta-error” if the agent is unable to find a skeletal plan that is able to handle this error.

The final approach for handling unexpected events was a limited ability to monitor the agent’s own progress. This monitoring was accomplished through generating expectations of results and comparing those expectations to the actual performance.

There was also a strongly distributed nature to the Phoenix architecture. The idea was that the same architecture would be used by a group of agents, one of which (in the simplest configuration) would act as a “boss” agent, and the others would perform a supporting role. Since the architecture requires that the agent have sensors and effectors, in a certain sense the supporting agents acted as sensors and effectors for the boss agent. In a more complex environment, one could have multiple boss agents each with a “sphere of influence” of agents reporting to it; the boss agents would then cooperate in
order to solve a problem. Although all agents have the same architecture (that is, they have a timeline, cognitive scheduler, plan library, state memory, sensors, effectors, and reflexes) they do not all have the same sensors, effectors. It is this difference that allowed the distinction to be made between boss agents and support agents. The lines of authority are clear to someone outside the system, although perhaps not clear within the system.

The Phoenix system was applied to a particular domain: fighting forest fires. The problem is to successfully fight a forest fire by surrounding it with firelines that are built by bulldozers. There is a fireboss, who is in charge of the fire-fighting process, and a number of bulldozers. Both the fireboss and the bulldozers are agents implemented using the Phoenix system. Such factors as the speed at which the fire moves, the size at which it grows, the wind, and the likelihood that it can cross a fireline are to some extent varied randomly so as to create a somewhat unpredictable domain. There is a set of skeletal plans for this domain. For example, for a relatively small fire, the skeletal plan might be for two bulldozers to start from the same place and to then encircle the fire in opposite directions until they meet.

There seem to have been a number of interesting ideas developed within the Phoenix system: the ability to respond to unpredictable events; the ability to behave intelligently in a real-time, distributed environment; the ability to use plans in a case-based reasoning type of approach. These all seem to have been relatively novel approaches at the time the system was developed. The biggest weakness that seems to exist in the Phoenix system is the inability to handle deadlines or utility of action in a meaningful manner. For example, if the fire were moving in the direction of a town, it might be desirable to prevent the fire from reaching that town at virtually any cost. This might involve the deployment of additional resources and of defending that town in particular. It isn’t clear that Phoenix can particularly reason about this type of situation. It would use the same plan as always for fighting a particular type of fire, without looking at how to prevent it from reaching the town.

Another weakness in the Phoenix system is that it would seem difficult for Phoenix to do effective replanning. If Phoenix constructs a particular plan based upon a skeletal plan, at what point is it to know that that skeletal plan is not working and it is time to go back to the drawing board and construct a new plan? Other planning systems have a well defined plan monitoring process where the expectation is raised that a certain amount of success in fighting the fire will have been attained by a certain time. If this doesn’t happen, it is automatically a flag to replan. In the case of Phoenix, there may be an action posted on the timeline to replan in response to certain events—such as the fire crossing the fireline—but there is no generalized monitoring process.
A more general monitoring process would look at when the crossing of the fireline is likely, for example, and would automatically replan in such a case, perhaps by reinforcing the fireline in such places. Still, the whole idea of using a skeletal set of plans in a planning system seems to be quite novel. This idea is likely to be very useful in domains such as aviation or medicine, where when one flies from one city to another there is a general plan that will achieve the goal which then needs to be tweaked for particular circumstances, or there is a general way of treating a particular type of patient that needs to be tweaked for the individual patient.

The system also is a good introduction to the use of multiple agents in a real-time problem. However, it seems to make a fairly strong assumption that the multiple agents are cooperating. It is not entirely clear how the ideas would be applied in a situation where the agents were achieving different goals, as for example in an aviation domain, or where they are hostile to one another as in a military domain. The different agents are assumed to be sensors and effectors for each other, and that is possible only in a cooperative situation. It is also not entirely clear how one would have the different agents change roles as became necessary by changes in circumstances. For example, if a fireboss sent a single agent to fight a particular fire, one of many fires happening at one time, and then that fire grew to the point where it became necessary to assign an additional agent(s) to fight it, then one of the agents would need to become a sort of second-tier fireboss. This would involve a change of role, and it is not clear how this would happen.

1.6 Book Synopsis

In the remainder of the book we explore the ideas described in this expository chapter in further detail. In the next chapter, we look at the formal conversion of planning into reaction—if we have a plan, then perhaps we can simply mechanically convert that plan into a reactive system. Some of the systems that we explore in Chapter 3 provide a language for expressing plans together with a formalism for converting the code written in that language into a reactive plan. We will argue, however, that this approach has certain limitations. It doesn’t take into account exactly how the agent is going to come up with the plan which it then converts into a reactive plan. Without taking this into account, every time a contingency arises the agent is reduced to first constructing a plan for dealing with that contingency and then reducing the plan to a reactive plan. In other words, the time taken will actually exceed that taken by just planning, which defeats the whole purpose of reaction.
In Chapter 4, we explore a possible solution to these issues: the idea of integrating planning and reaction. Rather than simply having planning and reaction be competing approaches that are the basis for an intellectual debate within the field, real systems should be built with the ability to integrate the best of the approaches into a single, hybrid system.

In order to do so, the agent must have a means for deciding whether to plan or whether to simply react. In Chapter 5, we describe an approach that one of us has developed for making this decision at planning time, and in Chapter 6 we describe an approach for making this decision at execution time.

Chapter 6 also describes various integrated real-time planning architectures that have been developed in the literature. In Chapter 7, we describe some extensions and applications of the approaches described earlier in the book.

Finally, in Chapter 8 we outline the strengths and limitations of the approaches described earlier in the book, and make a call to action to begin using the ideas from this book to better solve real-time, real-world problems in practice.

REFERENCES

Fikes, R., & Nilsson, N., "STRIPS: A New Approach to the Application of Theorem Proving to Problem

