Integrating Reactive and Deliberative Planning for Agents

Jim Blythe W. Scott Reilly

May 1993

CMU-CS-93-155

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Autonomous agents that respond intelligently in dynamic, complex environments need to be both reactive and deliberative. Reactive systems have traditionally fared better than deliberative planers in such environments, but are often hard to code and inflexible. To fill in some of these gaps, we propose a hybrid system that exploits the strengths of both reactive and deliberative systems. We demonstrate how our system controls a simulated household robot and compare our system to a purely reactive one in this domain. We also look at a number of relevant issues in anytime planning.

This work was supported in part by Fujitsu Laboratories, Ltd. and the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Fujitsu Laboratories or the U.S. Government.

Keywords: artificial intelligence, agent architecture, autonomous agents, reactivity, planning, problem solving, Prodigy, Oz

1 Introduction

Autonomous agents that respond intelligently in dynamic, complex environments, need to display a range of capabilities that have yet to be found in a single system. Deliberative systems that embody powerful techniques for reasoning about actions often fail to guarantee a timely response in time-critical situations [7]. Also, systems that respond well in time-critical situations typically do not provide a reasonable response in situations unforeseen by the designer [6]. An agent should be able to combine timely responses to well-understood situations with the ability to synthesize appropriate responses to novel situations. The performance of the agent should also degrade gracefully in situations that are increasingly novel and time-critical.

Reactive systems have traditionally been more successful than deliberative ones in controlling agents in the types of dynamic domains that interest us. They tend to be fast and designed specifically for controlling the execution of actions in dynamic environments. They also allow the builder to create idiosyncratic behaviors, which may be important in some domains or for creating more "realistic" behavior. Building a reactive system, however, is frequently a complex and time-consuming endeavor because of the need to pre-code all of the behaviors of the system.

Deliberative systems are best suited to long-term, off-line planning. This is effective for static environments, but not for controlling an autonomous agent which typically operates in a dynamic environment. Another strength of deliberative systems is their ability to synthesize plans which may interleave steps designed for multiple interacting goals from the relatively low-level input of domain operators. We have designed a hybrid reactive-deliberative system in an attempt to combine these complementary sets of capabilities in one system.

In this paper we describe our architecture, which integrates a reactive system and a deliberative planner that has been modified to be an instance of an anytime algorithm [3]. The reactive system uses hand-coded plans to react in situations foreseen by the agent builder and time-critical situations. When the reactive system has no appropriate pre-compiled plans or has extra processing time, it calls on the deliberative system. The two component systems and the integration are described in section 2.

Because we don't want to lose reactivity, we must arrange for the planner to be interrupted if necessary before it completes a plan. This is currently achieved by passing a fixed time bound to the planner, which solves as many goals as possible within the time bound before returning control to the reactive system. In order to do this, we modified the planner to be an anytime system, as described in section 3. Since we do not make any fundamental changes to the planner, we are able to take advantage of the body of work that has been done with classical planners, such as the use of abstraction [18], machine learning to improve planning performance [21, 11, 15] and derivational analogy [27, 16].

To give a feel for the type of behavior we have been able to get from our architecture, in section 4 we provide two traces of the system controlling a simulated household robot built in the Oz system [1]. In section 5 we present the results of some experiments we ran in the household robot domain as evidence that our hybrid agent compares favorably with an agent hand-coded specifically for this domain. In section 6 we discuss some work related to our own. Then, in section 7 we analyze some of the strengths, weaknesses, and tradeoffs in our architecture and in section 8 we discuss some avenues of future research.

2 The Architecture

We will first describe the reactive and deliberative systems and then describe how they work together.

2.1 The Reactive Planner: Hap

The Hap system [20] is designed to execute plans for achieving multiple, prioritized goals. It starts with a set of pre-defined goals and hand-coded plans for achieving those goals. The hand-coded plans are designed to allow Hap to respond to its environment in a timely manner.

Hap cycles through a set of decisions that concludes with the choice of an action to perform. The first step in each cycle is to choose a goal node to expand from the set of goals in the agent's *active plan tree* based on their *priorities*. There are three types of goal nodes: *acts* which represent physical actions, *mental-acts* which represent internal actions and *subgoals*.

If a mental-act is expanded, arbitrary lisp code associated with that mental-act is executed and Hap chooses another goal to expand. If an act is expanded, Hap stops processing and performs the chosen physical action. If a subgoal is expanded, Hap must next choose what plan to use to accomplish the goal.

Plans are chosen from the agent's *production memory*. These are rules whose *preconditions* test the current environment and the beliefs of the agent, and which specify sequential or parallel sets of goals to be added to the active plan tree. If multiple plans can be used, more specific plans are preferred to less specific ones. For instance, if an agent has a goal to go to the kitchen, a plan that is specifically designed to get to the kitchen will be chosen over a plan that gets the agent to any arbitrary room.

The language used in Hap is designed to make it easy for the user to define plans that are reactive. Two important constructs that facilitate this are *success tests* and *context conditions*. These change the status of the agent's goals and plans based on the agent's environment.

Success tests are sufficient conditions for goals to succeed. For example, a goal to clean a room can initiate a complex plan for dusting and vacuuming, but should another agent clean the room either before or during the execution of the plan, the goal should automatically succeed without the extra work of completing the plan. Whenever the success test of a goal in the active plan tree becomes true, the goal succeeds and Hap makes the appropriate updates to the tree.

Context conditions are necessary conditions for plans to be executed. A plan to sweep a floor is only appropriate as long as there is a broom in the agent's hand. If the agent drops the broom or another agent takes it away, it no longer makes sense to continue sweeping and other plans for achieving the goal should be examined. Whenever the context condition of a plan in the active plan tree becomes false, that plan fails and the tree is updated.

Hap also allows for demons that dynamically create new goals in appropriate situations. If such a goal has a higher priority than the other active goals, execution of the current plan is interrupted in favor of a plan to achieve the new goal. Hap will resume its previous plan once it has handled this unexpected event.

Hap is a descendant of Firby's RAP system [12] and is quite different from reactive systems like Brooks' subsumption architecture [5, 6]. The subsumption architecture uses S-R rules for driving action directly from the sensed environment. Hap is a reactive system in the sense that it uses pre-defined plans with reactive annotations (the success-test and context-conditions) to achieve its goals. This means that Hap uses explicit representations of goals and plans, which is important for our purposes as it allows for meaningful communication with the deliberative system.

2.2 The Deliberative Planner: Prodigy

Prodigy 4.0 [2] is a classical deliberative planner that uses means-ends search to create plans from descriptions of operators, given initial and goal state descriptions. A Prodigy plan is a linearly ordered sequence of operator steps. Given a set of goals described in a typed first-order logic, Prodigy repeatedly selects a goal, an operator and a set of bindings so that the operator's effects will unify with the goal. Preconditions of the operator that are unmatched in the current state are then added to the set of goals. Prodigy produces plans that achieve multiple goals simultaneously. Goal statements as well as the preconditions of operators may be arbitrary expressions of first-order logic, involving existential and universal quantification.

The search that Prodigy conducts involves a number of choice-points, and Prodigy uses *control rules* to represent information about them. These choice-points occur when it selects a goal to work on, an operator to achieve the goal when there are multiple ways to proceed, and bindings for the operator when different objects can be used. Control rules can also suspend search paths and move to different ones. The control rules are if-then rules whose left hand sides can access information about the current state of the world as well as the state of the search process.

Prodigy does not (yet) represent uncertainty or non-deterministic operators, or interleave plan execution with plan synthesis[1]. By default, Prodigy does not produce more than one plan to achieve a goal, although it can represent a plan as a partial order [26].

2.3 Integration

Hap is designed to react quickly and intelligently in a dynamic environment by using stored behaviors when possible. Prodigy is designed to plan for sets of goals that may interact, and to learn to plan more effectively. We integrate these two systems so as to retain the strengths of each by giving primary control of the agent to Hap. Hap keeps the tree of goals and plans that the agent is pursuing up to date. When there are stored plans available or there is a strict time constraint, Hap will usually act in a pre-programmed way.

When Hap has extra time to act or there is no stored plan to handle the current situation, Hap may call Prodigy. Planning is considered a behavior like anything else Hap does[2], so the conditions under which Prodigy is called are

[1] This constraint is relaxed in Gil's work on learning by experimentation [14].

[2] It is actually achieved with a Hap mental act. See section 2.1.

contained in the pre-defined Hap productions. This allows the agent builder to create agents that deliberate more or less as desired. The call to Prodigy includes a time bound, so that the agent does not lose reactivity even though planning may take an arbitrary amount of time to complete.

When Hap calls Prodigy, it passes a predefined subset of the agent's goals. Because Hap and Prodigy have somewhat different notions of what goals are, not all Hap goals are expressible as a Prodigy logic expression. However, all of the goals in the active plan tree that can be so expressed are passed to Prodigy for planning.

Whereas Hap only pursues one goal at a time, Prodigy, as a complete planner, can create an interleaved plan that achieves a number of goals simultaneously. This allows it to cope with resource contention and to produce more efficient plans. Prodigy takes the set of goals with priority information and a time bound, and uses time-dependent planning techniques to come up with the best plan it can in the given time. This means that the plan may not solve all of the goals but, if it solves any, it will solve the ones with higher priorities. Time-dependent planning is discussed in section 3.

The integration works, in part, because the communication between Hap and Prodigy occurs at an appropriate level of abstraction, both of state and operators. Like other researchers [13, 24], we have used an abstraction boundary between the reactive and deliberative components of our architecture to address one of the major problems facing planning systems in dynamic worlds: a planning system must invest a certain amount of time to create a plan even though that plan is based on assumptions that change over time. For this reason, deliberative planners typically assume a static or near-static world. While this allows the planner to construct a plan, in a dynamic domain this plan will frequently fail, because some of its underlying assumptions have become false since the plan was constructed. This is one of the primary reasons deliberative planners have been unsuccessful as agent architectures.

Instead of changing the static-state assumption that the planner makes, we can improve the quality of the plans by making the planner plan in a more static state. Since the environment cannot be easily changed, we instead provide the planner with an abstraction of the state that tends to include the static elements and exclude the dynamic ones, and allow Prodigy to plan in that state.

Examples of the static elements of a state are the location and connectivity of rooms and furniture. Examples of the dynamic elements of the state are the location of other agents and the fact that a given door is open or closed.

So, Prodigy looks at the world with many of the dynamic elements filtered out and produces plans that rely only on the more static elements of the state. In addition it is possible to pass to Prodigy a notion of how rapidly various domain facts change, for example in terms of a Markov process, and have the planner reason about this information while constructing a plan. This is a current research topic in Prodigy that is not expanded upon in this paper.

Prodigy uses the abstract state to generate abstract plans. By an abstract plan, we mean a plan that is a sequence of Hap subgoals instead of a set of concrete actions. Hap uses its set of stored reactive plans for executing these subgoals in the dynamic world. Hap is able to fill in the dynamic details that Prodigy did not plan for. Also, Hap will often have numerous alternative plans for achieving a subgoal, so a single Prodigy plan can generate very different behavior depending on the current state of the world.

This process might work as follows[3]: our robot agent, Mr. Fixit, is in an environment as pictured in figure 2. Hap generates the two goals of recharging the robot's battery and dusting the bedroom. Hap passes an abstraction of the state to Prodigy that looks like this:

```
(and (in recharger closet)
     (in fixit kitchen)
     (dirty bedroom))
```

Obviously, much of the state has been left out, like the connectivity of rooms and the location of a number of movable objects. Prodigy uses this state and generates the following plan to achieve the two goals: ((goto closet) (recharge) (goto bedroom) (dust bedroom)). Each of the operators in the plan, goto, dust, and recharge, are Hap subgoals, and Hap's production memory contains reactive plans for achieving these goals. These plans are designed to be interrupted and restarted in most cases and typically have contingency plans for various problems that may be encountered. They also take into account the dynamic elements of the world. So, where Prodigy just plans to (goto closet), Hap deals with, for example, unlocking and opening doors along the way.

The actual level of abstract communication between Hap and Prodigy is not uniquely determined. We could have passed information about the connectivity of the rooms to Prodigy as well as the state that we did. We made the decisions about what pass to Prodigy by hand to best fit our domain. There

[3] This is a simplified example.

may be some automatic ways of helping decide what to include in the abstract state, but generally we expect these to be domain-dependent decisions.

3 Time-dependent Planning

We aim to allow agents built with Hap to construct plans without sacrificing reactivity. A necessary condition for this is that Prodigy be able to respond within a bounded time period — sometimes with less time than it normally needs to complete a plan. If the planner is unable to suggest at least some action in these situations, the agent may become frozen, unable to make a reasoned response to its environment.

A number of different types of algorithms have been proposed for time-dependent problems. Fixed-time algorithms always use the same amount of time to compute their output, regardless of their input. Variable-time algorithms are given a time bound as a separate parameter, and will produce outputs of different qualities for different time bounds. Anytime algorithms [3] are not given a time bound when started, but can be stopped at any time, and should return a reasonable output whenever stopped.

We have modified Prodigy to be an instance of an anytime planning algorithm, but in this paper we use it as a variable-time algorithm by having Hap pass a time bound when it calls Prodigy. The reason for this is that although an anytime algorithm may sometimes be more useful, Hap and Prodigy must be run concurrently in order to use it, and this has not yet been implemented.

3.1 Anytime Planning in Prodigy

Prodigy is made an anytime planner by slightly modifying the algorithm described in section 2.2. The basic idea is to keep track of the best state encountered while planning, where the goodness of a state is defined to be the sum of the utilities of the top-level goals that are achieved in the state. Here is the modified version of the basic planning algorithm:

Prodigy is given a conjunction of goals, a set of operator schemas and control rules as input. These top-level goals are provided with demons, which will force a new state to be compared against a "best" state whenever one of the goals becomes true. The set of active goals is initialized to the top-level goals.

With the demons and initial state score in place, Prodigy begins to plan as normal. It repeatedly chooses a goal from the set of active goals, and an operator to achieve the goal. If the operator can be applied to the current state, it chooses whether to do so or delay application. This choice enables it to solve nonlinear problems (see [26] for details). If it chooses to apply the operator, the state is updated appropriately. It may then choose to apply other operators whose application have been delayed or choose another goal and repeat.

Whenever on operator is applied, the values of some state literals are changed. If the truth value of a top-level goal is changed, the demon adjusts the current state value by its goal's utility. Since the state score is changed incrementally, maintaining the score has no significant cost. If the score is the best so far, the new state and score are saved along with a pointer to the sequence of operators that produces them.

If Prodigy is interrupted before it completes, it returns the plan for the current best state along with the state and its score. This allows Hap to determine how good the partial solution is. In our current implementation the interrupt will occur after a fixed time-bound, but this is not necessary. When we have Hap and Prodigy running as concurrent processes, Hap will be able interrupt Prodigy at any time and this algorithm will still work.

If Prodigy completes a plan before an interrupt is signaled, it returns the plan immediately. In future implementations we will consider ways to improve on the quality of the plan before the interrupt is signalled, although Prodigy will still signal to Hap that a complete plan is available. We discuss this further in section 8.

Note that this modified algorithm does not take significantly more time or space than the original Prodigy algorithm to find a complete plan. The extra time to initialize the demons is linear in the number of top-level goals, and the extra cost of tracking the best state is small compared with the cost of the search. The space required to save the best current plan is generally smaller than the size of the final plan returned. This algorithm is much better able to deal with problems where Prodigy is not given enough time to produce a complete plan, however.

3.2 Discussion

Much of the work to date on anytime planning systems has simplified the planning task significantly. In [4], Dean and Boddy state that "most useful anytime algorithms we know of apply to sufficiently simple problems that any interesting planning problem will require combining the results of several anytime algorithms". However, it is the essence of nonlinear planning problems that there is no guarantee that combining the results of planning for individual

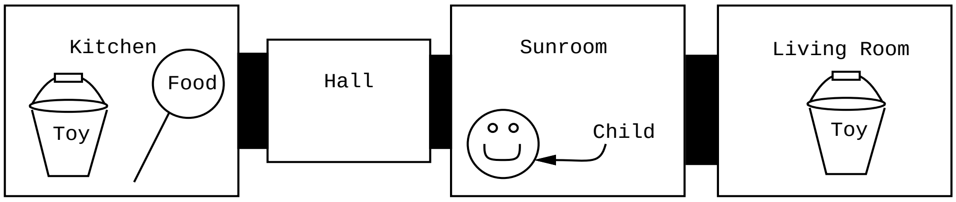

Figure 1: Simple planning domain.

subproblems will yield a solution to the set of subproblems. We have developed an anytime planner that can handle tasks as complex as any complete, nonlinear deliberative planner.

As an illustration, consider the following scenario. A child has the goals of playing with a toy and eating some food. Suppose the initial world is set up as shown in figure 1. There are toys in the living room and kitchen, and the food is in the kitchen.

Using a standard anytime decision procedure as described in [4], the child might set up two separate processes, one for the goal of playing with a toy and one for the goal of eating food, and then decide how much time to allocate to each. If the goals have equal utility and are allocated roughly equal amounts of time, the procedure dealing with the goal of playing with a toy will return first, with the plan to go to the living room. If the child follows this plan, it will be unable to discover the optimal plan, for which it need only go to kitchen. If the child gave the conjunction of its two goals to a deliberative planning system, it would not return until it had solved both goals and might still suggest moving to the living room first.

In our modified algorithm, however, what happens depends on the amount of time allocated to the planner. The planner first picks one of the goals to work on, say the goal of playing with a toy. When this is solved, it plans for the goal of eating some food. If the time limit runs out before the second goal is solved, it returns a plan to move to the living room and play with the toy. The child would then have to re-invoke the planner with the second goal at some later stage. If time allows, the planner will also solve the second goal and come up with the same plan as the other methods, namely to go first to the living room and then to the kitchen. However, if the deadline is still not met, the system keeps searching for a better solution, and given enough time will find the optimal solution of heading directly to the kitchen.

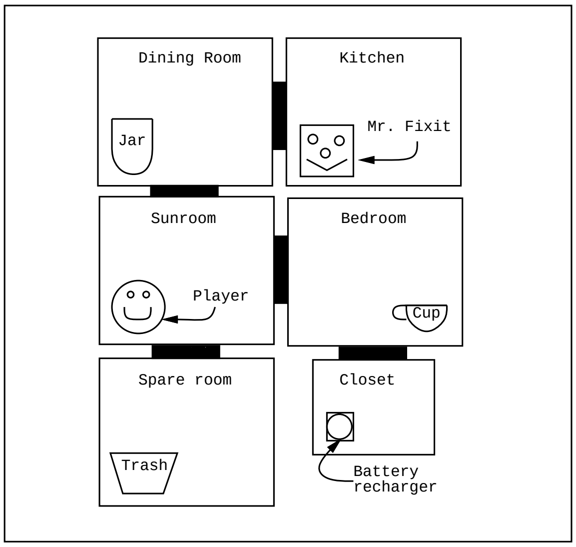

Figure 2: Rough layout of the simulated household robot environment

4 A Household Robot: Mr. Fixit

4.1 Traces

We have designed and built a simulated household robot agent, Mr. Fixit[4], using this architecture. Mr. Fixit lives in a simulated environment built with the Oz system [1]. Figure 2 gives a rough layout of the initial environment that Mr. Fixit inhabits. Many of the details of the world have been left out as they are unimportant for the behaviors described here[5]. The *player* is a human user that is also interacting with the simulation.

Figures 3 and 4 show two traces from runs of this simulation. The traces have been edited for brevity and clarity. The information in the traces is that of an omniscient third party. Typically the user will control the player by typing text commands and receive sensory information about the world as natural language descriptions. The simulation allows each agent to choose an action in a round-robin fashion and keeps time in discrete units.

Mr. Fixit begins the simulation with a model of the world created by walking through the world once. However, given that Mr. Fixit has limited sensing abilities and there is an unpredictable user in the simulation, his model

[4]This example was inspired by [25].

[5]There are roughly 85 objects in the simulation.

```
Fixit:     *SAY  "I am beginning my rounds."
            *GO-TO  Dining-Room
            *GO-TO  Sunroom
            *SAY  "Hello."  to Player
            *GO-TO  Spare-Room

Player:  *GO-TO  Spare-Room

Fixit:     *SAY  "Hello."  to  Player
            *GO-TO  Sunroom
            *GO-TO  Bedroom

[Plan: goal: (recharge battery)
         time limit: 1 second
         plan: ((goto closet)
                  (recharge))]

Player:  *TAKE  Trash-Can
Fixit:     *GO-TO  Closet
Player:  *GO-TO  Sunroom
Fixit:     *RECHARGE  Fixit  using  Battery-Recharger
Player:  *DROP  Trash-Can
Fixit:     *GO-TO  Bedroom
Player:  *GO-TO  Bedroom
Fixit:     *SAY  "Hello."  to  Player
Player:  *BREAK China-Cup

[Plan: goal (thrown-out-trash)
         time limit: 1 second
         plan: ((goto cup)
                  (get cup)
                  (goto trash-can)
                  (put cup trash-can))]

Fixit:     *TAKE  China-Cup
            *GO-TO  Sunroom
            *PUT  China-Cup  in  Trash-Can
```

Figure 3: Sample trace of the household robot

of the world will often be incomplete or incorrect.

The robot has a number of important goals that define its behavior. They are, from most to least important: recharge battery when low, clean up broken objects, greet the player, dust dirty rooms, and roam the house looking for tasks to perform. Demons determine when the goals become active — for example, the goal to greet the player is only active when Mr. Fixit comes across the player.

In figure 3, Mr. Fixit begins roaming the house, as there is nothing else that needs to be done immediately. When he encounters the player in the sunroom, he interrupts his plan to roam the house and greets the player. He then returns to his previous behavior. Fixit's rounds eventually take him to the bedroom. While in the bedroom, Fixit notices his battery running low and calls Prodigy. Prodigy returns a plan within the given time limit of one

```
Player:  *BREAK  China-Cup
Fixit:     *GO-TO  Sunroom

[Plan: Goal: (and (recharge)
                        (thrown-out-trash))
          Time limit: 1 second
          Plan: ((goto recharger)
                    (recharge)
                    (goto cup)
                    (get cup)
                    (goto trash-can)
                    (put cup trash-can))]

Fixit:     *GO-TO  Bedroom
             *GO-TO  Closet
             *RECHARGE
             *GO-TO  Bedroom
             *TAKE  China-Cup
             *GO-TO  Sunroom
             *GO-TO  Spare-Room
             *PUT  China-Cup in Trash-Can

[Plan: Goal: (clean-dining-room)
          Time limit: 1 second
          Plan: ((goto dining-room)
                    (dust dining-room))]

Fixit:     *GO-TO  Sunroom
             *GO-TO  Dining-Room
Player:  *GO-TO  Sunroom
Fixit:     *DUST  Dining-Room
Player:  *GO-TO  Dining-Room
Fixit:     *SAY  "Hello."  to  Player
Player:  *BREAK  Jar
```

```
[Plan: Goal: (and (clean-dining-room)
                            (thrown-out-trash))
          Time limit: 1 second
          Plan: ((goto jar)
                    (get jar)
                    (goto trash-can)
                    (put jar in trash-can)
                    (goto dining-room)
                    (dust dining-room))]

Fixit:     *TAKE  Jar

[Plan: Goal: (and (clean-dining-room)
                            (thrown-out-trash)
                            (recharge))
          Time limit: 0.5 seconds
          Plan: ((goto recharger)
                    (recharge))]

             *GO-TO  Sunroom
             *GO-TO  Bedroom
Player:  *GO-TO  Sunroom
Fixit:     *GO-TO  Closet
             *RECHARGE

[Plan: Goal: (and (clean-dining-room)
                            (thrown-out-trash))
          Time limit: 1 second
          Plan: ((goto trash-can)
                    (put jar trash-can)
                    (goto dining-room)
                    (dust dining-room))]

Fixit:     *GO-TO  Bedroom
             *GO-TO  Spare-Room
             *PUT  Jar  in  Trash-Can
             *GO-TO  Sunroom
             *SAY  "Hello."  to  Player
             *GO-TO  Dining-Room
             *DUST  Dining-Room
```

Figure 4: Sample trace of the household robot

second and Fixit begins executing the plan.

In the meantime, the player moves the trash can to the sunroom and then goes into the bedroom and breaks a china cup. Fixit has finished recharging and notices the broken cup. Again, Fixit calls Prodigy and it returns a plan for disposing of the cup. This plan is generated under the faulty assumption that the trash can is still in the spare room. When Fixit goes to execute the plan, however, he notices the trash can in the sunroom and is able to adapt without re-planning.

In figure 4, Mr. Fixit is already in the spare room performing rounds while the Player is in the bedroom. The player breaks the cup. Fixit notices the broken cup at the same time that his battery needs recharging. Prodigy is notified of the two goals and is able to find a plan to solve them both within one second. Fixit executes the plan without any problems. During the execution of the plan Fixit goes by the Player twice without stopping for the traditional greeting. This is because the current plan being executed has a higher priority than greeting the user.

Once the two goals are accomplished, Fixit notifies Prodigy of a goal to clean the dining room, which is planned for successfully. During the execution of this plan, the player enters the dining room and is greeted because dusting is a low priority goal. Despite the warm greeting, the Player decides to break the jar. Fixit notices this and plans to clean up the mess and then return to dusting. While cleaning up the broken jar, Fixit's battery again gets low. The three pending goals are sent to Prodigy along with a half second time limit[6]. Prodigy is able to solve only the goal to recharge in this amount of time and Fixit executes that plan successfully. Once that is done, Fixit calls Prodigy to replan for the other two goals and executes them.

4.2 Discussion

In this section we look a little deeper at what's behind the design of Mr. Fixit. One of the main concerns in building a hybrid architecture is how quickly Prodigy is able to plan for goals in the domain. If Prodigy isn't given enough time to generate plans for even single goals, then the agent is going to be stuck.

On the other hand, if we know that Prodigy is going to generally be given more than enough time to achieve some subset of goals, then there are tradeoffs in speed vs. plan interleaving. For example, say Fixit knows about 2 broken objects that need to be thrown out. If Prodigy only has time to solve one of the goals, it will return a plan that may be sub-optimal with respect to solving both goals.

		Time bound (seconds)			
		0.25	0.50	1.00	2.00
Number of goals	1	0.90	0.98	1.00	1.00
	2	0.80	0.96	1.00	1.00
	4	0.38	0.52	1.00	1.00
	8	0.13	0.16	0.62	1.00

Table 1: Average proportion of goals solved.

We tested Prodigy's ability to solve goals quickly in this domain by giving it problems to solve with varying numbers of top-level goals and varying time

[6] The time bound was set by hand, but could also be set by Hap.

bounds. The results of the test are reproduced in table 1. The table shows the average number of goals Prodigy was able to solve out of increasingly large goal sets. We can see how increasing the complexity of the problem affects the amount of time Prodigy needs to come up with a complete solution. Larger goal sets make more complex problems because of the increased chance of goal interactions and the greater number of decisions the planner needs to make.

For example, the table shows that when Prodigy has two goals to solve and a 0.5 second time bound, it is able to solve an average proportion of 0.96 of the goals — or 1.92 goals on average.

When Prodigy is given a 2 second time bound, it can always solve the goal conjunct in this relatively simple domain, making the time-dependent planning redundant. With smaller time bounds, Prodigy is only able to solve for a subset of the goals. If Prodigy only returned complete solutions Hap wouldn't get any useful information in these time-critical cases.[7]

This information can be used to guide the design of the domain specification given to Prodigy. If we know that Prodigy will often only have 0.25 or 0.5 seconds with which to work, we will want to generate simple serial plans. If the domain allows Prodigy to take 2 or more seconds, we can write control rules for Prodigy that will produce more efficient plans but that will usually take longer before completely planning for any single goal.

5 Experiments: Deliberative+Reactive vs. Reactive

We used the household robot domain as a testing ground for our architecture. We have already discussed some of the obvious benefits to using a deliberative planner as part of an agent architecture, but if the architecture doesn't perform well, these benefits may be overshadowed. To evaluate our architecture we decided to test it against a purely reactive agent. This agent was written entirely in Hap and was designed specifically for this domain. In general, the hand-coded plans were similar to the ones that Prodigy generated, but we also added specific interleaved plans for throwing out multiple pieces of garbage. We did not expect the hybrid agent to do quite as well as the reactive agent, but if our architecture could come close, then we can reap the benefits of using

[7]This data was collected on a Hewlett Packard 720 workstation. The 50 test problems for each data point were randomly created with repetitions removed. To be more complete we could have also collected information about how often specific goals tend to arise in the domain and how long it takes to plan for each goal type (e.g. planning to dust a room is generally easier than planning to throw out trash).

a deliberative architecture without concern for losing in other areas.

The domain described in section 4 was modified as follows. First, we changed the procedure for throwing out garbage so that the robot first had to get a bag out of a cabinet in the kitchen, then put the trash in the bag, then put the bag in a trash bin in the sunroom. Second, a second robot, the Destructo2000 was added to the environment. This robot would generate cups and break them on the floor with some probability that we could control. Third, a (slightly unrealistic) phone was placed in the bedroom. With a 10% chance the phone would ring during any turn it was off the hook. If the robot hadn't answered the previous call, it was lost. Until another call came in, the previous caller would keep ringing. Fourth, all the rooms started in need of dusting and didn't become dirty again once dusted. Fifth, the player was removed from the simulation. Sixth, the new goal priority ordering was (from most to least important): recharge battery, answer phone, throw out trash, and dust rooms.

The phone and the battery created occasional interrupts in behavior that were kept constant over every run. We were able to control how dynamic the environment was by changing the chance that the Destructo2000 would drop a cup. We ran both the hybrid and pure reactive robots with this chance at 5%, 8%, and 10%. We also ran the hybrid system at 3%. The data presented represent the results over 10 runs for each robot-environment pair.

The hybrid system was set up to allow Prodigy a 3 second time bound. Although we didn't do as complete an analysis of this version of the domain as was described in section 4.2, some informal experiments showed that the plans required to handle the new trash procedure required enough time that 3 seconds was not enough to consistently create interleaved plans for throwing out garbage. Because of this, the hybrid agent always generated serialized plans for throwing out trash. These were sub-optimal plans, but Prodigy was always able to solve at least one goal in the allotted time.

Figure 5 graphs how well Mr. Fixit did in keeping up with the Destructo2000. It's fairly clear that at 5%, 8%, and 10%, Mr. Fixit is falling further and further behind and eventually will be swamped with cups. At 3%, however, Mr. Fixit is able to keep up with the dropping cups.[8]

The purely reactive agent that we created had hand-coded plans for throwing out multiple pieces of trash simultaneously. Because of this we expected

[8]Note that the number of cups represents the total number of cups on the floor over 10 runs of the simulation.

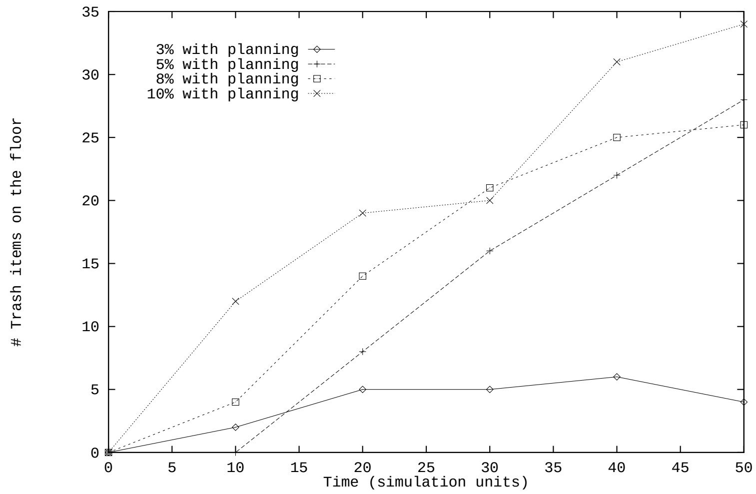

Figure 5: Hybrid robot cleaning up cups in dynamic domain.

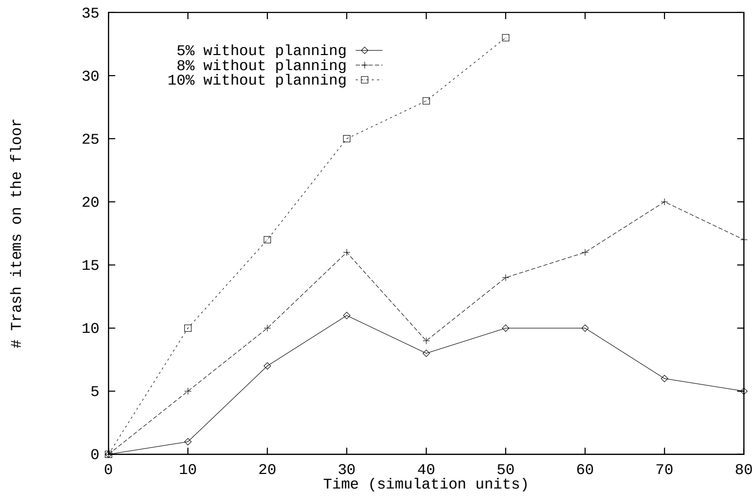

Figure 6: Pure reactive robot cleaning up cups in dynamic domain.

the reactive agent to perform better in cleaning up after the Destructo2000. Our expectations were realized and figure 6 shows how at 10% the reactive agent is falling behind, but at both 8% and 5% the robot seems able to keep up.

The relative performances of the two systems in picking up cups was expected. The reactive system, which was designed to handle that particular task more efficiently, did better. That the hybrid system fared as well as it did on this task is reassuring. Admittedly, this is still a relatively simple domain so differences in performance will tend to be small, but at the same time moving to more complex domains will make it even harder for builders of reactive agents to create a compete set of behaviors for all situations in a reasonable amount of coding time.

What was somewhat surprising about our results was how well the hybrid system did on the other tasks it was given. First, both agents performed at 100% on battery recharging, which was the most important goal the agent was given. Second, the hybrid agent was able to answer the phone at a rate of 78% as compared to 61% for the reactive agent. Third, figure 7 shows how well the two agents did at dusting the rooms. As expected, in the more dynamic domain, the agents had less time to clean rooms, but the hybrid agent was able to do almost as well as the reactive agent in the 10% domain and even slightly better in the 5% domain.

A plausible explanation for the relative performances on the battery recharging, phone answering, and room cleaning tasks is that the reactive system tended to spend more time between the kitchen and the sunroom picking up cups. Meanwhile, the hybrid agent was more likely to be in other areas of the house, especially the bedroom and the closet, which made answering the phone and recharging take less time. In other words, we expect that this is probably a product of the domain and not attributable to our architecture.

Finally we consider the time taken by the agents. Each time cycle in the simulations with the reactive agent took 9.2 seconds. This includes both agents, the physical world simulation, and the data gathering. When we changed to a hybrid system, this increased to 9.7 seconds. This means that on average, the hybrid agent took only 0.5 seconds longer to choose an action than the reactive agent. Furthermore, because of the time bound, the time used in planning never exceeded 3 seconds for any turn. Finally, because Hap controlled when Prodigy was called and for how long, it could have be designed to specifically not call Prodigy or to call Prodigy with a small time bound in some circumstances (e.g., a low battery) in order to ensure that the

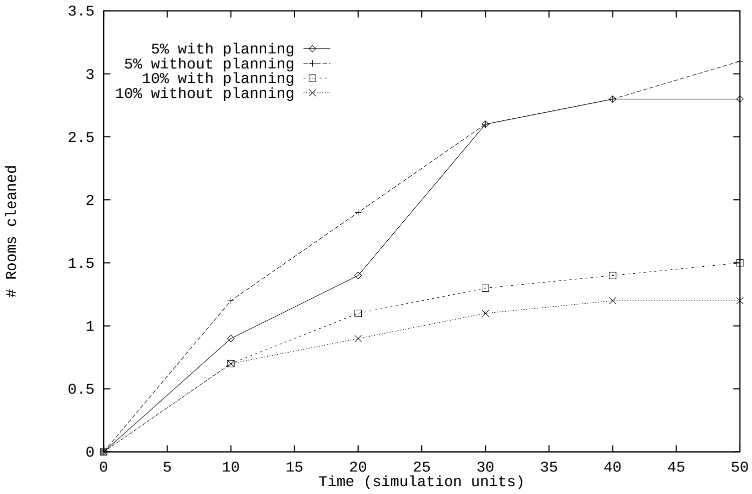

Figure 7: Robots cleaning rooms in dynamic domain.

agent acted quickly in time-critical situations. As we were more interested in maximizing deliberation in this agent, we chose not to make such design choices.

6 Related work

The two distinct parts to this work, the architecture and the anytime planning algorithm, both have relations to other work in the field.

6.1 Architecture

Our work is similar in spirit to that of Payton, Rosenblatt, and Keirsey [22]. They have designed a two-level architecture where the high-level planner generates plans for navigating an autonomous land vehicle that are then executed by a reactive low-level system. This work differs from ours in that they are primarily concerned with path planning and we are concerned with complete, general-purpose planning.

The deliberative/reactive split with planning at an abstract level is closely related to the work by Erann Gat [13] on the ATLANTIS system. That architecture is somewhat different in that the reactive system is subsumption-based and there is a separate controller module (which, like Hap, is based on Firby's RAP system) that switches control between the two levels. We combined the reactive and controller modules within the reactive level. Gat also had an extended deliberative level, which performed tasks such as stereo vision processing, but had a scaled-down planner. We chose to use a full planning system. Finally, Gat's domain was simple enough that he didn't have to be concerned with anytime planning at the deliberative level.

The Sepia architecture for planning and learning [24] also takes a very similar view to our own. The major differences appear to be that their reactive architecture is subsumption-based and they use a different method of anytime planning (that of Elkan, discussed in the next section).

6.2 Anytime Planning

Elkan [9] has proposed an anytime planning algorithm that works by gradually relaxing assumptions until the plan is correct according to the planner's knowledge. If this system is stopped before completing its task, it will return a plan that is not necessarily correct. Our method on the other hand will return a correct plan for a subset of the goals or a less than optimal plan for all of the goals. This is preferable if the goals have roughly independent utilities and do not clobber each other, and this is generally the case in the examples we have studied.

Drummond and Bresina's ERE system [8] uses an anytime planning technique that is based on forward projection. As they admit, the search space rapidly becomes very large for realistic problems, and would require extensive control knowledge. Our use of means-ends analysis as well as control knowledge helps reduce the search space.

Washington and Hayes-Roth [28] and Hendler [17] have also studied the use of abstraction as a means of dealing with time bounds. We plan to investigate using Prodigy's existing abstraction mechanism in our current system. This will provide additional planning granularity in situations where not even one subgoal can be fully planned for.

Russel and his colleagues [23] and Korf [19] have investigated tradeoffs in the search mechanisms to reason about time bounds effectively. This work on search can be viewed as complementary to our own, since one must still face the eventuality that not all goals will be solved in the time allowed or all interactions fully accounted for within the time bound. Prodigy is able to make use of these search algorithms.

7 Conclusions

Let us now take a step back and review what our goals were in developing this architecture and analyze how well this particular system satisfies those goals.

We don't believe that either purely reactive or purely deliberative systems are sufficient for controlling autonomous agents in dynamic environments. Subsumption-based reactive systems don't have the internal representations to display long-term, goal-driven behavior to any degree. Reactive systems like Hap solve this problem, but are still hard to code in that all situations must be foreseen by the agent builder and goal interactions must be specifically accounted for. For example, the reactive robot described in section 5 had to be designed to interleave plans for throwing out multiple cups at once. Even in a domain as simple as this, accounting for more complex goal interactions would be extremely difficult. Despite these drawbacks, reactive systems have generally been fairly successful because of their ability to execute in dynamic situations. Reactive systems also allow designers to create agents with idiosyncratic behaviors that are more interesting or realistic where that is desirable.

On the other hand, deliberative systems are better at planning for interacting goals, but don't generally perform well in dynamic domains because they assume the environment is static. We also want to draw on the significant research that has been done with learning, abstraction, and derivational analogy in deliberative systems. So far, we have not taken advantage of much of the work based on deliberative planners, but as our system is almost unchanged from the normal Prodigy system, we are optimistic that this will be feasible.

The hybrid architecture we built provides a good deal of power to agent builders and we expect this to be useful in designing agents for rather different types of domains. For example, in highly dynamic domains where quick action is vital, the agent builder can put reactive behavior to deal with most situations into Hap and design the Prodigy system to return quickly. This will often be at the expense of plan quality, because the interactions between goals will not be explored. In more stable domains where Prodigy is given greater amounts of time to plan, the plans will tend to be much more efficient than those created by a reactive system. In fact, in some situations it might be the case that Prodigy is able to solve some resource critical problem that a reactive system might not solve at all.

Our architecture is not, however, without weaknesses. Our current model of anytime planning only gives credit to states that solve one or more top-level goals. If progress can be made towards a top-level goal but no such goal can be completely solved within the time bound, the planner will be unable to suggest an action. An analysis of the domain, such as that described in 4.2 will give a feel for the ability of the deliberative system to handle goals within specific time bounds, but this is not foolproof and occasionally the planner may be stymied.

A further problem that is not directly related to the architecture we have chosen is that the stochastic nature of plan execution means that the planner cannot know the exact duration of many of the operators in its plan and is therefore unable to make a tight schedule. Solving this problem will require rough models of duration that can allow the planner to schedule for the worst case, or probabilistic models that can be attributed a probability of success.

8 Future Work

We have presented a framework for integrating deliberative and reactive planning for autonomous agents. The initial system already improves the abilities of agents to react to unforeseen situations in a dynamic world. There are several enhancements we are considering for the next version of the integration.

First, Prodigy currently plans afresh each time the planner is called. Hence, if called with the same problem, it will only get further the second time due to the effects of leaning and case-based reasoning. While this will give some advantage, we would like to have Prodigy directly build on the search trace it previously produced.

In order to do this, Prodigy needs a notion of *plan update* and *plan extension* to determine which parts of the search are still valid after the state has changed and how to best fix those parts that are no longer valid. We expect this to have a strong overlap with the replay mechanism of Prodigy's case-based system, except that in the paradigm we discuss here the previous plan may not be complete.

Second, It might be possible to make the planning system more efficient if it were given a fixed time bound and knowledge about the expected time to solve a set of goals given the state and the agent's operators. (Dean et al. [4] use this information in a technique called "deliberation scheduling") We wish to generate such knowledge for our planner from experience. This same type of scheme should allow Prodigy to automatically learn when to solve goals in a serial manner and when to attempt to interleave interacting goals.

To achieve this, simply keeping statistics on the time to plan for various goals would not be effective, since this time depends heavily on a number of factors unrelated to the goal, including: the other goals that may interact, the number and types of objects in the planning domain and the planning state. To produce useful planning time knowledge, one might build the categories for which the statistics are kept at the same time as we gather the statistics. This task has been studied by Etzioni [10] and we aim to take a similar approach. This could involve using explanation-based learning to build the categories about which we keep empirical data.

Third, as these agents exist in dynamic environments with incomplete and incorrect knowledge, we are interested in the question of plan robustness. There are at least two distinct avenues to explore here: expanding the role of Prodigy and creating a new type of learning.

The first approach is to extend the Prodigy planning paradigm to allow plan enhancement and contingency planning. If there is extra time after a plan is generated, Prodigy should be able to use that time to reason about ways to make the plan more robust. This would include using more reliable operators, creating contingency plans, and relying only on more stable aspects of the state. Second, we also hope to explore some learning mechanisms that will enable Prodigy to come up with more robust plans based on a better understanding of the dynamics of the environment.

9 Acknowledgements

We would like to thank Jaime Carbonell, Joseph Bates and the rest of the Prodigy and Oz groups for their multi-faceted assistance. This work is supported in part by Fujitsu Laboratories, Ltd. and the U.S. Air Force Avionics Laboratory.

References

[1] Joseph Bates. Virtual reality, art, and entertainment. *PRESENCE: Teleoperators and Virtual Environments*, 1(1):133–138, 1992.

[2] Jim Blythe, Oren Etzioni, Yolanda Gil, Robert Joseph, Alicia Pérez, Scott Reilly, Manuela Veloso, and Xuemei Wang. Prodigy4.0: The manual and tutorial. Technical report, School of Computer Science, Carnegie Mellon University, 1992.

[3] Mark Boddy. *Solving Time-Dependent Problems: A Decision-Theoretic Approach to Planning in Dynamic Environments.* PhD thesis, Brown University, May 1991.

[4] Mark Boddy and Thomas Dean. Solving time-dependent planning problems. In *International Joint Conference on Artificial Intelligence*, 1989.

[5] Rodney Brooks. Intelligence without representation. In *Proceedings of the Workshop on the Foundations of Artificial Intelligence*, June 1987.

[6] Rodney Brooks. Integrated systems based on behaviors. In *Proceedings of AAAI Spring Symposium on Integrated Intelligent Architectures*, Stanford University, March 1991. Available in *SIGART Bulletin*, Volume 2, Number 4, August 1991.

[7] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–378, 1987.

[8] Mark Drummond and John Bresina. Anytime synthetic projection: Maximizing the probability of goal satisfaction. In *AAAI*, 1990.

[9] C. Elkan. Incremental, approximate planning. In *National Conference on Artificial Intelligence*, 1990.

[10] O. Etzioni. Embedding decision-analytic control in a learning architecture. *Artificial Intelligence*, 49(1-3):129–159, January 1991.

[11] Oren Etzioni. *A Structural Theory of Explanation-Based Learning.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1990. Available as technical report CMU-CS-90-185.

[12] James R. Firby. *Adaptive Execution in Complex Dynamic Worlds.* PhD thesis, Department of Computer Science, Yale University, 1989.

[13] Erann Gat. On the role of stored internal state in the control of autonomous mobile robots. *AI Magazine*, 14(1):64–73, Spring 1990.

[14] Yolanda Gil. *Acquiring Domain Knowledge for Planning by Experimentation.* PhD thesis, School of Computer Science, Carnegie Mellon University, August 1992. Available as technical report CMU-CS-92-175.

[15] John Gratch and Gerry DeJong. Composer: A probabilistic solution to the utility problem in speed-up learning. In *AAAI 92*, 1992.

[16] Kristian J. Hammond. *Case-based Planning: An Integrated Theory of Planning, Learning and Memory.* PhD thesis, Yale University, 1986.

[17] James A. Hendler. Abstraction and reaction. In *AAAI Spring Symposium on Planning in Uncertain, Unpredictable or Changing Environments*, 1990.

[18] Craig A. Knoblock, Josh D. Tenenberg, and Qiang Yang. Characterizing abstraction hierarchies for planning. In *National Conference on Artificial Intelligence*, 1991.

[19] Richard E. Korf. Real-time heuristic search. *Artificial Intelligence*, 1990.

[20] A. Bryan Loyall and Joseph Bates. Hap: A reactive, adaptive architecture for agents. Technical Report CMU-CS-91-147, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, June 1991.

[21] Steven Minton. *Learning Effective Search Control Knowledge: An Explanation-Based Approach.* Kluwer, Boston, MA, 1988.

[22] David W. Payton, J. Kenneth Rosenblatt, and David M. Keirsey. Plan guided reaction. *IEEE Transactions on Systems, Man, and Cybernetics*, 20(6):1370–1382, November/December 1990.

[23] Stuart Russel and Eric Wefald. *Do the Right Thing.* MIT Press, 1991.

[24] Alberto Segre and Jennifer Turney. *Planning, Acting and Learning in a Dynamic Domain*, chapter 10. 1993.

[25] Lee Spector and James A. Hendler. Knowledge strata: Reactive planning with a multi-level architecture. Technical report, University of Maryland, College Park, 1990.

[26] M. M. Veloso. Nonlinear problem solving using intelligent casual-commitment. Technical Report CMU-CS-89-210, School of Computer Science, Carnegie Mellon University, 1989.

[27] Manuela M. Veloso. *Learning by Analogical Reasoning in General Problem Solving.* PhD thesis, Carnegie Mellon University, august 1992.

[28] Richard Washington and Barbara Hayes-Roth. Abstraction planning in real-time. In *AAAI Spring Symposium on Planning in Uncertain, Unpredictable or Changing Environments*, 1990.