Advances in Cognitive Systems (2015) 1-16 Submitted 2/2015; published

An Architecture for Flexibly Interleaving Planning and Execution

Yu Bai YBAI181@AUCKLANDUNI.AC.NZ
Chris Pearce CPEA144@AUCKLANDUNI.AC.NZ
Pat Langley PATRICK.W.LANGLEY@GMAIL.COM
Mike Barley MBAR098@CS.AUCKLAND.AC.NZ
Charlotte Worsfold CWOR015@AUCKLANDUNI.AC.NZ
Department of Computer Science, University of Auckland, Private Bag 92019, Auckland 1142 NZ

Abstract

In this paper, we present a theory that aims to reproduce behavioral abilities that humans use to generate and execute their plans. We begin by highlighting the phenomena we are interested in, and then present several theoretical claims that account for them. Next, we introduce FPE, a five-stage system that supports strategies for flexible execution, including open-loop and closed-loop control. We then turn to the extensions we have made both to this system and to FPS, a flexible problem solver, in order to reproduce a range of interleaving strategies. We report runs with the architecture that support our claims for its coverage and flexibility. We also analyze how these different techniques perform in response to variations in domain characteristics. In closing, we discuss related work in these areas and consider avenues for future research.

1. Introduction

Humans exhibit great flexibility in how they carry out complex activities. In some cases, they pay close attention to the environment and their actions, engaging in 'closed-loop' behavior. In other cases, they act in a more automated manner, not bothering to check whether their actions' conditions are met or their effects are produced, relying on 'open-loop' control. These two methods represent opposite ends of a behavioral continuum; strategies towards the 'closed-loop' end of the spectrum prioritize the cost of error over the cost of sensing, while those towards the other end do the reverse.

We observe similar variability in how humans interleave planning and execution. In some cases, they generate an extended plan before they begin to carry it out; in others, they behave more reactively, putting little thought into the future before they act. Many factors appear to influence such choices, from availability of information to environmental predictability, but the flexible character of planning and execution is an important feature of human cognition.

The AI planning community has reported various strategies for interleaving planning with execution, but those systems have been optimized for certain contexts, rather than designed to support flexible strategies. We desire a theory that can support the variability observed in humans and, to this end, we have adopted five theoretical assumptions. In this paper, we describe these high-level claims and introduce an integrated system that instantiates them. We begin by describing our execution module, FPE, in terms of the hierarchical plans it carries out, its five-stage process, and the strategic knowledge that produces its behavior. Then we briefly review FPS (Langley et al., 2013), an architecture for flexible problem solving that we use to generate our plans. Next, we introduce the additional strategic knowledge we have devised for both FPE and FPS to support flexible interleaving. Having described the integrated system in its entirety, we present a set of experimental results for a number of domains and discuss our findings. We conclude by reviewing related work and discussing our plans for future research.

2. Behavioral Abilities and Theoretical Claims

The overall purpose of this work is to account for the variety of ways that humans carry out their plans. We have identified two key behavioral abilities on which to focus:

- *Humans can utilize different techniques for executing complex plans.* For instance, in some situations they pay close attention to the effects of their actions, but in others they simply assume that they are successful.
- *People also have at their disposal a diverse range of techniques for moving back and forth between plan generation and execution.* They might generate a complete plan and then carry it out; move frequently between the two process; or adopt an approach that falls somewhere between these two extremes.

Although we are especially interested in the second phenomenon, we believe that a truly flexible theory of planning and execution should reproduce both of these abilities. To this end, we adopt five high-level theoretical claims:

- Plans are represented by trees, in which each node is a problem, and every child denotes a subproblem of its parent.
- The process that enacts these plans operates in a loop of five discrete stages: intention selection, condition checking, intention enaction, perceptual inspection, and effects checking.
- A separate cycle is responsible for producing these plans. This too involves five stages: problem selection, intention generation, subproblem generation, failure checking, and solution checking.
- Domain-independent strategic knowledge governs decisions at several stages of each cycle to produce a range of execution and planning strategies.
- In addition to influencing how complete plans are generated and executed, strategic knowledge also determines when the planner transfers control to the executor, and vice versa.

In the sections that follow, we introduce FPE, an architecture for flexible execution that carries out hierarchical plans, such as those produced by FPS (Langley et al., 2013). We have also made extensions to both FPE and FPS so that they can interact with one another in a flexible manner. Together, the integrated system incorporates all of the postulates described above.

We believe that reproducing the flexible execution and interleaving behaviors exhibited by humans is a valuable goal in its own right. However, an additional benefit of our integrated system is that it lets us test hypotheses about interactions between strategies and domain characteristics — such as the reliability of the agent or the environment — in a common infrastructure. We will return to this functionality in Section 5, where we present experiments that test these interactions.

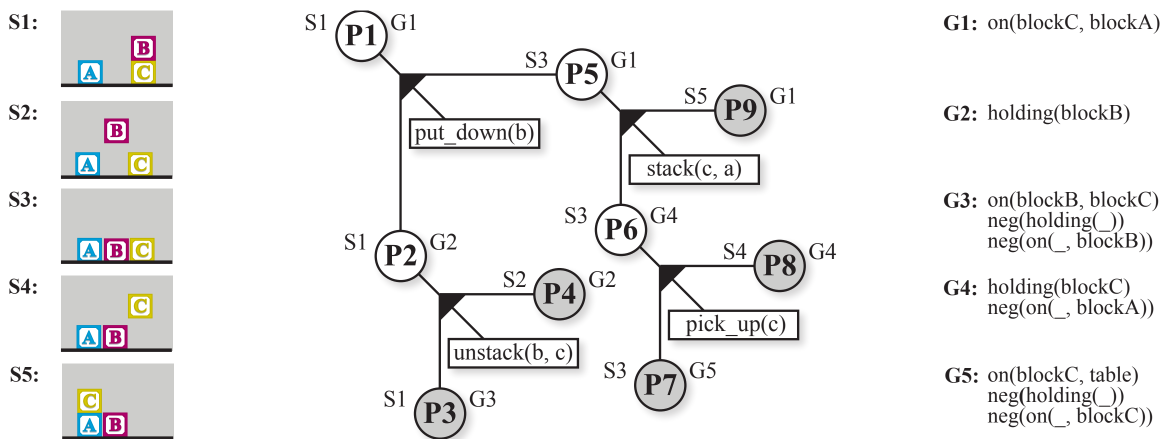

Figure 1. An example plan from the Blocks World domain. Each nontrivial problem decomposes into two subproblems with an intermediate intention, specifying a hierarchical solution that involves four actions: unstack block B and put it on the table, then pick up block C and stack it on A.

3. A Framework for Plan Execution

We will focus first on the execution of complex plans. For the purposes of this paper, we will assume these plans are produced by FPS, but this approach should apply equally well to the outputs of other planning systems. After reviewing the structure of plans, we discuss FPE, our system for flexible plan execution, in terms of both the control architecture and the knowledge it employs.

3.1 The Hierarchical Structure of Plans

Plans generated by FPS are hierarchical and comprise two key elements. The first of these is the *problem*, which has an associated *state description* and *goal description*; these descriptions have unique identifiers so they can be used elsewhere in the plan. If a problem's goal description is satisfied by its state description, then we say that it is *trivial*, which means that the problem is solved.

The second key element is the *intention*, which denotes a specific instance of a domain operator, including its instantiated conditions and effects. An intention's conditions describe a set of elements that must be true for that intention to be applicable. These elements take the same form as the predicates in a problem's state description — for instance, `on(blockA, blockB)` indicates that block A must be on top of block B. An intention's effects specify the changes that it will make to the current state if it is enacted.

If a problem, P, is nontrivial, it can be associated with one or more intentions, I, each of which breaks it into two subproblems: a *down subproblem* that shares P's state but has goals based on I's conditions, and a *right subproblem* that has the same goals as P but a state that results from applying I to P's state. A decomposition of P is a solution to P if each subproblem is either trivial or has its own pair of solved subproblems.

Figure 1 illustrates a simple plan that should clarify this organization. This example is from the Blocks World domain (Fikes & Nilsson, 1972), in which an agent must rearrange a collection of blocks on a table so that they match a particular configuration. The state description of the initial problem specifies that blocks A and C are on the table and block B is stacked on top of C. The goal description simply stipulates that C should be on top of A. The plan shown in the figure is just one of many possible decomposition trees that will solve this task.

The initial problem, P1, is nontrivial, so it can be broken into a down subproblem, P2, and a right subproblem, P3, by the intention to "put down block". P2 takes the intention's single condition — that the gripper is holding block B — as its goal. Since this element is not included in its state description, P2 is nontrivial like its parent. Therefore, it is attached to two subproblems via an intention. These subproblems, which result from the "unstack block B from C" intention, are both trivial, so they do not have decompositions of their own and represent a solution to P2.

P3's state description results from the effects of both "unstack block B from C" and "put down block" on P2's state. Like its parent, P3 has a solution that consists of two intentions: "pick up block C" and "stack block C on A". All of the terminal nodes in the tree are trivial, which means that it is a solution for P1. Figure 1 clarifies the hierarchical nature of FPS's plans. This organization incorporates our first theoretical postulate: that plans are stored as problem trees, in which every node (except the root) is a subproblem of its parent. This has implications for the plan execution module, to which we now turn our attention.

3.2 The Execution Process

To carry out the hierarchical plans described above, we have developed FPE, a flexible execution module that operates in discrete cycles. This system incorporates our second claim from Section 2, which states that execution should be distinct from planning, but, like that process, should involve five stages: intention selection, condition checking, intention enaction, perceptual inspection, and effects checking. To clarify this procedure, we consider each stage in turn and describe how it might carry out the initial intention of our example plan.

In the first stage, *intention selection*, FPE must choose which intention in the plan to carry out next. This stage typically involves little choice. However, if an intention was not successfully executed in the previous cycle, then the system may choose to reselect it. For example, suppose that we want it to execute the solution shown in Figure 1. Before the execution process begins, we simply pass the module the entire tree. Upon entering the first stage of its cycle, the system finds the first intention in that plan, namely, "unstack block B from C".

Once FPE has made this selection, the next stage, *condition checking*, involves deciding whether the intention is applicable in the current state of the world. The system may choose to analyse the *current world description*, an internal state description that encodes the system's beliefs about the world (as opposed to the real external environment in which the system acts). When the system reaches this step in our Blocks World plan, it follows its default behavior and matches each condition against the state elements in its current world description. In doing so, the module makes two assumptions: that its environment is fully observable and that the intention is only applicable if all conditions are met. Nothing has altered the environment between plan generation and condition checking, so the module determines that the intention is still applicable and marks it as such.

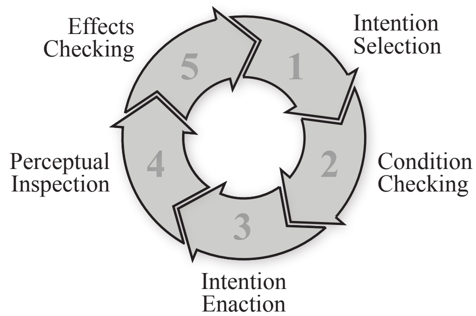

Figure 2. The five stages of FPE's execution cycle.

If the conditions are satisfied, or if the module makes that assumption, then it enters the *intention enaction* stage, and carries out the associated action.[1] As in problem selection, this stage involves little choice. At this point in its execution of our Blocks World example, the module attempts to carry out its intention to "unstack block B from C". This may occur in either the physical world or a simulated environment. Let us assume that, in this case, FPE is unsuccessful and that the effects of the action are not applied to the simulated environment.

It will then begin *perceptual inspection*, which involves acquiring information about the new environmental state and updating FPS's current world description. This ensures that the current world description does not fall out of step with FPE if an action fails, or an agent or unexpected event alters the external environment. Since the gripper failed in its task at the previous stage, the current world description remains the same: block B is still on C and the gripper is empty.

Effects checking is the final stage of execution, and it is at this point that FPE determines whether the applied intention produced its intended effects. If they have been produced, or if the system assumes that they have, then it updates its current world description to reflect this. By default, unsuccessful effects will lead FPE to reselect the intention and try it again in the next cycle, unless the number of attempts has already exceeded a limit. At this point in our example, the system matches the expected effects of its current intention against the current world description. It expects to find that the gripper is holding block B, but this is not the case. It does not mark the intention as successful and, thus, may reselect it and try again at the next intention selection stage.

In summary, FPE's execution process involves five discrete stages. The module may select an intention, check whether the current state satisfies the intention's conditions, carry out the selected action, perceive the new state of the world, and ascertain whether the appropriate effects occurred. To produce flexible execution, our system supports alternative behavior at three of these stages. We shall now describe this functionality in detail.

1. In our current work on FPE, we assume that actions are discrete and that the system knows when it has completed an intention (regardless of whether it was successful or not).

3.3 Flexible Plan Execution

As noted earlier, humans exhibit considerable variability when executing complex plans and procedures. When errors are expensive, they can be very careful; a pilot, for example, pays close attention to conditions and effects during takeoff and landing. In contrast, people carry out many procedures, like taking a shower, on autopilot, devoting their attention to other matters. These extremes, sometimes referred to as *closed-loop* and *open-loop* control, are just two execution strategies at our disposal. They highlight the fact that a strategy's suitability depends partly on the cost of execution errors, although other factors, such as the reliability of actions and the cost of perception, are also relevant. Recall that this is the first behavioral ability that we noted in Section 2.

To support such flexibility, FPE utilizes strategic knowledge to determine what choices it makes and what elements it checks in working memory. This domain-independent content takes the form of control rules that refer to meta-level predicates like *problem*, *state*, *goal*, *intention*, *condition*, and *effect*. These *strategic control rules* can influence the architecture's behavior at the second, fourth, and fifth stages.

- At the second stage, the module might check the intention's conditions against the current world description, or it may simply assume that the conditions are satisfied without bothering to check that this is the case;
- Strategic control rules for the fourth stage determine whether FPE senses the environment to collect information about its state, or if it moves on to the next stage without doing anything;
- In the final stage, the module can either ensure that the intention's effects have been applied to the world, or it can simply assume that it was successful and move on.

Although the control schemes described above are individually simple, they can interact to alter behavior substantially. In the previous section, we described how the module might begin to execute the blocks world plan from Figure 1. In that example, the system checked that the state of the environment met the current intention's conditions, perceived the state of the external environment, and checked that the effects of the intention have occurred. Now, imagine that it is executing the same plan, but strategic knowledge specifies that it should do nothing at these three stages.

In this scenario, FPE begins as it did before and selects the first intention in the plan. During the second stage, it does not check the conditions of this intention; instead, it simply updates working memory to indicate that the intention's conditions are satisfied. The module then moves straight to the intention enaction stage and attempts to "unstack block B from C". As before, it fails in its task. However, since it does not perceive the environment or check that the expected effects have occurred, the module remains oblivious. It simply assumes that the intention was successful, and marks it as such. Therefore, when the system reaches the next intention selection stage, it moves on to the "put block B down" intention. It fails to carry out the action again because, this time, the intention's conditions are not met. FPS continues in this way until it reaches the end of its fourth cycle, at which point it ends the execution process having not enacted any intentions successfully.

Decisions regarding the use of execution strategies primarily involve analyzing the speed and accuracy required when acting in a domain. A plan that can be executed in an open-loop manner is likely to maximize execution speed, but this comes at a cost in accuracy when acting in volatile settings. Unexpected changes in the world can lead to the plan being unsuccessful without the

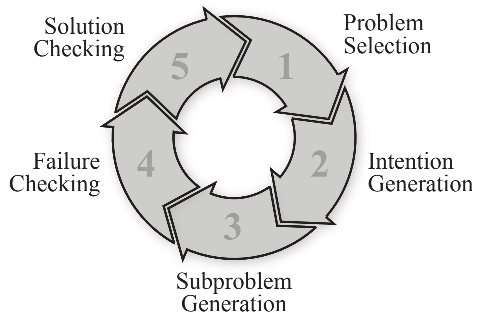

Figure 3. The five stages of FPS's problem-solving cycle.

executing agent being aware of its failure until after an entire sequence of actions. In contrast, closed loop behavior involves repeatedly examining the environment during plan execution. This takes more attentional resources but also increases the chances of success in unpredictable contexts.

4. Flexible Interleaving of Planning and Execution

A complete agent architecture should support not only execution and planning, but also their integration. In this section, we briefly review FPS's five-stage problem-solving process and the knowledge that modulates it. We then discuss how we have extended both the planning and the execution process to incorporate our fifth claim, which states that strategic knowledge should govern the transfer of control from one module to the other.

4.1 FPS's Planning Process

Any discussion of our combined framework's interleaving process necessitates an understanding of FPS's problem-solving cycle. Therefore, we will provide a brief overview of how it generates the hierarchical plans described earlier. As in the execution module, this system loops through a set of five stages, in accordance with our third postulate from Section 2. Figure 3 depicts the names of these stages and the order in which they occur. Strategic control rules, similar to those in FPE, let the system solve problems in different ways. Although the examples that we discuss in this paper all refer to planning domains, the FPS system may also perform different varieties of problem solving, such as design and theorem proving.

In the first stage, *problem selection*, the system picks a problem to focus on. At the outset, only one alternative is available, but as the initial problem is decomposed recursively into subproblems, FPS has more options. Next, during *intention generation*, the system finds operator instances relevant to the current problem. The third stage, *subproblem generation*, involves selecting an intention and using it to decompose the current task into subproblems. After this, *failure checking* detects issues that may lead FPS to abandon the current problem, then *solution checking* determines whether any more work is required to solve it. If it finds at this final stage that it has not satisfied its initial goals, then FPS continues for another cycle of problem solving.

FPS can call upon strategic knowledge at each stage to produce behavior. Rules for problem selection determine whether it uses depth-first search, iterative sampling, breadth-first search, or some other search regimen. Those for intention selection govern whether FPS carries out forward search or means-ends analysis. During subproblem generation, strategic knowledge provides domain-independent heuristics to evaluate intentions based on the number of their conditions that are not met, the goals they achieve if applied, or some other control scheme. Finally, strategic control rules for failure checking specify various criteria, such as loop-triggered failure, or a particular depth-limit.

Taken together, strategic control rules let FPS reproduce a broad range of problem-solving strategies that have appeared in the cognitive science literature. For instance, to reproduce depth-first means-ends analysis, the system adopts depth-first search to select problems, backwards chaining during intention generation and loop-triggered failure.

4.2 Strategic Knowledge for Interleaving Planning and Execution

Just as agents can exhibit different strategies for execution, so too can they employ different strategies for interleaving execution with planning. Any account of such variation should cover the variety of behaviors observed in humans and machines. In some settings, one can generate a complete plan before executing it in an open-loop manner. In others, the problem is so complex, as in some difficult puzzles, or the environment is sufficiently unpredictable, as in playing chess, that one must alternate between extending a plan and executing it. Execution can occur under different conditions, for example, whenever one solves a down problem, or after completing N-step lookahead. The same holds for planning, which can occur frequently (e.g., whenever one makes a move) or rarely (e.g., only when an executed plan does not go as intended). These different approaches relate to the last behavioral ability we discussed in Section 2. Recall that this relates to the broad range of strategies that humans employ to move from planning to execution and vice versa. We will not attempt to enumerate all possibilities here, but we will return to this concept later.

Both FPS and FPE play central roles in our account of these variations and, as before, differences in domain-independent strategic knowledge are responsible for producing different behaviors. The primary loci of control reside in the fifth stage of problem solving — solution checking — and the second stage of execution — condition checking. This functionality incorporates our fifth theoretical claim: that domain-independent strategic control rules determine whether the system proceeds to the next stage of the current process or transfers control to the other module.

Strategic knowledge for the solution checking stage of the planning process encodes four alternative stopping criteria. The first and simplest criterion, *full solution*, passes control to execution when FPS finds a complete plan for the initial problem. The remaining criteria let FPE take over as soon as the planner has found a partial solution: *solved subplan* requires FPS to solve a single executable subproblem; *long enough plan* generates a sequence of *N* intentions that the agent can carry out in the current world state; and *enough goals satisfied* produces a subplan that achieves *N* percent of the initial goals. As soon as FPS discovers that it has satisfied its stopping criteria, it sends the solved problem and its associated plan to the execution module. FPE then immediately enters the intention selection stage of its cycle, and attempts to carry out the first step.

We also implemented three stopping criteria for the second stage of the execution cycle. These control schemes are responsible for transferring control from execution back to problem solving, and, thus, determine how much of the current plan to execute. The strategic knowledge for the first stopping criterion, *execute as much as possible*, shifts processing back to FPS when FPE has either carried out its entire plan or that plan has failed. Alternatively, strategic control rules might specify the *enough execution* scheme that returns control to planning if FPE has carried out *N* intentions; or the *enough goals achieved* criterion that does so if it has produced a state that satisfies a particular percent of the initial problem's goals. Once it regains control, the planner begins in the problem selection stage.

We can clarify these criteria with an example from a domain we call *Robot Messenger*, which was inspired by the work of Haigh and Veloso (1998). In this domain, K different rooms are connected by K hallways, and they are arranged so that a robot can travel from one room to any other through a single hallway. Suppose that the robot starts in one of three rooms, B (which is also the location of the key to room C), and that the mail is in the locked room C. The only goal specifies that the mail must be delivered to room A.

For this example, assume that the FPS problem solver adopts the strategy of means-ends analysis with the *long enough plan* stopping criterion (with *N* set to three), and the FPE execution module utilizes the *execute as much as possible* criterion and reactive control. The planner uses means-ends analysis to recursively generate decompositions that satisfy at least one of their parent problem's goals. After decomposing a number of problems, FPS discovers that it has found a solution that involves three intentions. Recognizing that it has satisfied the *long enough plan* criterion, FPS gives control to the execution module during the success checking stage. FPE then uses closed-loop control to carry out the three actions. Upon reaching the success checking stage, FPE realizes that it has satisfied its execution stopping criterion, so it returns control to the planner.

The joint framework continues unimpeded until the robot unlocks the door to room C. FPS creates a three step plan that involves entering room C, picking up the mail, and moving from room C to hallway 3. However, once FPE has carried out the first two actions, the robot accidentally drops the letter before leaving the room for the hallway. After transferring control back to the planning module, it creates a new plan that involves returning to room C to pick up the mail, and then taking a different hallway to reach room A. The next planning and execution cycles proceed as intended and the robot successfully completes its plan by depositing the mail in room A.

This is just one of the many approaches to interleaving execution and planning that FPE and FPS jointly support. However, it should, clarify how the two modules interact and suggest how other strategies could modify processing in different situations.

5. Experience with the Integrated System

Now that we have described our integrated system for generating and executing plans, we can report our experience with it. In Section 2, we discussed the abilities that we wish to support. These high-level aims suggest two hypotheses about our system's behavior: that it supports a variety of strategies for plan execution that it reproduces a range of techniques for interleaving execution with planning. In this section, we present empirical evidence for each hypothesis in turn.

Table 1. Descriptions of five domains used in testing execution strategies.

BLOCKS WORLD. This domain contains *N* blocks, each of which can be located either on top of another block or on the table. A gripper can pick up and put down blocks that do not have anything on top of them. It cannot pick up more than one block at a time. The goal description of a problem describes a partially specified configuration of blocks.

DOCKWORKER. Piles of containers are scattered about this domain; next to each pile is a crane, which can load the top container onto a robot and vice versa. A single operator lets this robot transport a single container from one location to another. Goal descriptions specify the desired locations of specific containers.

LOGISTICS. This domain involves a number of packages, which can be transported by truck between two locations within a city, or by plane between two cities. Goal descriptions specify the final locations of packages.

ROBOT MESSENGER. In this domain, a robot must deliver *N* mail items to specific rooms. To do so, it must navigate a system of hallways and rooms, which can be connected by one or more doors; if a door is locked and the robot does not possess a key, or if a hallway is blocked, then the robot cannot enter and must find another way. Goal descriptions specify the room to which each item of mail must be delivered.

TOWER OF HANOI. In this domain, *N* disks of varying sizes sit on three pegs. It only takes a single action to move a disk to a new position, but a disk can only be placed on either an empty peg or a larger disk, and can only be moved if there is nothing on top of it. Goal descriptions describe an arrangement of disks.

5.1 Flexible Plan Execution

To demonstrate our first behavioral target, we created the knowledge required to encode the five domains listed in Table 1, as well as ten problems for each of them. These domains include two classic puzzles — Blocks World and Tower of Hanoi — a planning domain — DockWorker — and two transport domains — Logistics and Robot Messenger. Additionally, we have implemented each of the four execution strategies described in Section 3.3: open-loop execution without monitoring conditions or effects, closed-loop execution where both conditions and effects are checked, a strategy where only conditions are checked, and finally a strategy with only effect checking.

In order to provide an environment for tests of plan execution, we connected FPE to a simulated environment with which it could interact. Within this simulated environment, actions were considered to be executed instantaneously, and FPE was able perceive a complete description of the state of the world at any time. Our simulated environment also let us alter the likelihood of actions being successful. For each of the five domains, we provided FPE with complete plans to each of the ten defined problems and made it execute these plans within the simulated environment with the probability of action failure set at 20%. We performed these runs once for each of the above execution strategies, measuring the number of plans correctly executed in the simulated environment.

- The first execution strategy we have supported is purely *open- loop control*, which does not acquire or use any external feedback to determine whether the environment meets its expectations. In FPE, this strategy results from doing nothing at the second, fourth, and fifth stages of the execution cycle; the module simply selects an intention and then immediately executes it.
- At the other end of the spectrum is *closed-loop control*, which uses information from the environment to ascertain whether an intention's preconditions are satisfied and whether it has had the expected effects. This schemes lets the module adjust its execution process, for example, by reselecting and enacting a failed intention. This strategy involves active processing at the stages for precondition checking, perceptual, and effects checking.
- In addition to these two established approaches to physical control, we also used FPE to implement two hybrid strategies. The first utilizes precondition checking and perception, but does not check an action's effects. This strategy should do well in domains where the agents are very likely to succeed, but where the environment is also likely to be disturbed by external events.
- The final strategy utilizes perception and effects checking, but does not check preconditions. This combination seems appropriate for domains that would involve a stable environment that is unlikely to be affected by external events, but in which the agent's actions are not entirely reliable.

Closed-loop execution was the most successful of these strategies, correctly executing 92% of the plans it was provided. Open- loop execution only completed its plans in 18% of the runs, while the condition-checking and effect-checking strategies completed plans in 30% and 90% of their runs respectively. The value of execution strategies such as the ones presented naturally depend on the costs of checking conditions, effects, and perceiving the world in a given scenario. Although our experiment does not consider these costs, we have demonstrated that our system can use several different execution strategies.

5.2 Interleaving Planning and Execution

We now turn to the ability of the integrated system, its support of different strategies for interleaving planning and execution. Our integrated framework provides us with a rare opportunity to investigate interactions between strategies and domain characteristics and we have designed an experiment to study them. For this experiment, we added functionality to the simulated environment that lets it introduce random events between the execution of operators. For instance, in our Robot Messenger domain, the agent might accidentally drop the letter it is carrying. Such occurrences change the simulated world in unexpected ways and often necessitate re-planning.

We implemented five different strategies for interleaving planning and execution. These strategies emerge from the combination of control knowledge for both the planner and execution system, as well as knowledge that determines when control should pass from one process to the other. We hypothesize that, in general, the system's success rate will decrease as the probability of unexpected events occurring in the simulated environment increases. We also expect that, within this overall trend, strategies that plan further ahead will be less successful than those that turn to execution as soon as they identify helpful actions.

- The first strategy was *simple open-loop execution of a complete plan*. One can question whether this qualifies as interleaving, since planning runs to completion before FPS transfers control to FPE, which then carries out the plan without examining the environment. Nevertheless, this scheme involves both planning and execution, and serves as valuable strategy for comparison with more intricate techniques. To produce this strategy, we employed the *full solution* option in FPS and the *execute as much as possible* stopping criterion in FPE, along with no precondition checking, perception, or effects checking.
- Our second strategy involved *closed-loop execution of a complete plan with recovery*. Here the integrated system generates a complete problem solution through planning, before executing as much of that plan as possible. At any point, if FPE recognizes that it can no longer execute its plan, the module will give control back to FPS, which then generates a revised plan based on the environmental state. To model this strategy, we utilized *full solution* in FPS and *execute as much as possible*, precondition checking, perceptual inspection, and effects checking in the FPE module.
- Next, we produced a *one-step forward search strategy*, which involves minimal forward planning and is, therefore, at the other end of the spectrum from complete plan strategies. The system passes control to the execution module once it has found a plan with one applicable intention. As soon as it has executed that action, FPS takes over again to replan. In addition to closed-loop control, this strategy adopts the *long enough plan* stopping criterion for FPS — creating plans that include just one intention — and the *enough execution* criterion for FPE — enacting a single intention before returning to planning.
- We also implemented *closed-loop execution of subplans as soon as they have been completed*. This technique offers a compromise between focusing on the end goal and keeping search tractable, as well as modulating the extremes of purely open-loop and purely closed-loop execution. We implemented this strategy by invoking the *solved subplan* stopping criterion for FPS and the *execute as much as possible* stopping criterion and closed-loop control for FPE.
- Finally, we produced *three-step lookahead with single-step execution*, a strategy that mimics the behavior of many game-playing systems. This repeatedly carried out three-step forward search followed by a single execution step. To this end, we invoked forward chaining with three-step lookahead during FPS's planning process, along with single-step execution, precondition checking, perceptual inspection, and effects checking during FPE's execution cycle.

For each domain, we ran the integrated system on each of the ten defined problems. We did this five times, starting at a 0% chance of an event occurring after the execution of an action and increasing this probability by 10% after each complete problem set. We repeated this entire process five times in total to deal with the system's non-deterministic processes, and recorded the number of problems solved within a within a 6,000 planning cycle limit.

Table 4 shows the results of runs that were performed within the Tower of Hanoi domain. For each interleaving strategy, points are plotted for rate of events to show how many problems the system successfully completed within the given cycle limit. For all strategies except one-step lookahead, as the rate of unexpected events increased, the combined system found it more difficult to complete problems within the allocated cycle limit. One-step lookahead's performance understandably did not vary greatly; even with no events, only one action would be planned before execution

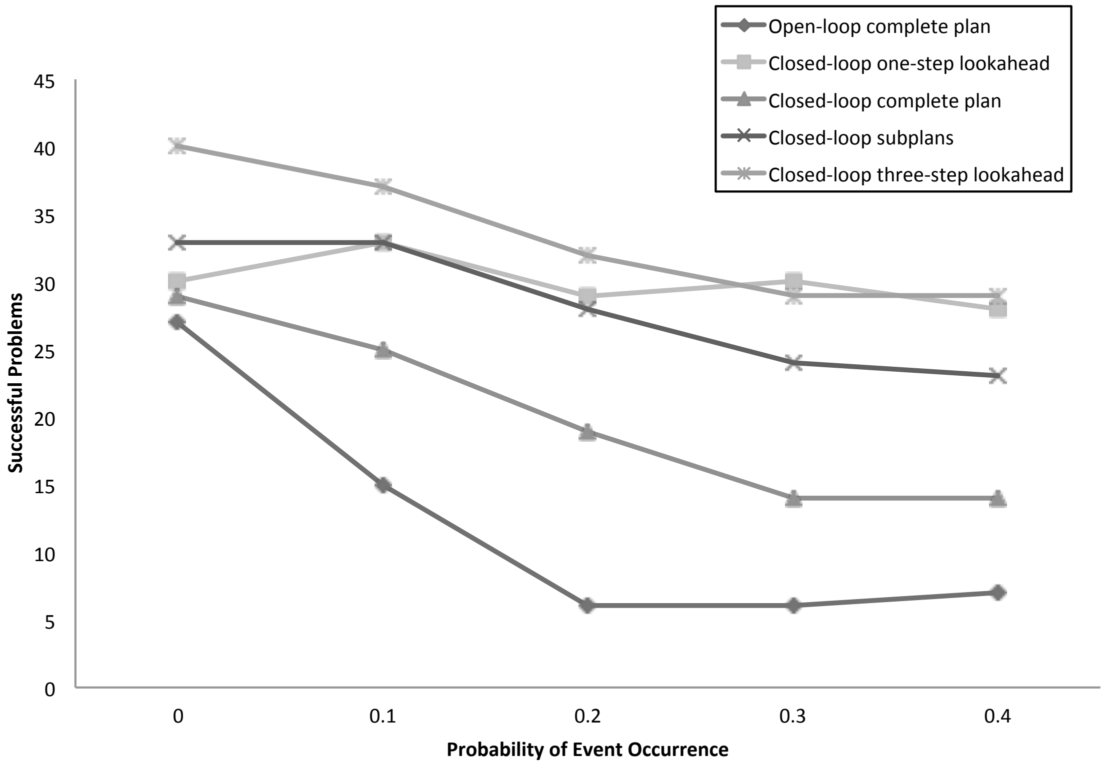

Figure 4. The success rate of our five interleaving strategies as the probability of random events increases. This graph represents the performance of the system on a single Tower of Hanoi problem

was performed and the changes to the world were perceived. Additionally, there is a clear difference between the performance of strategies that created full plans before passing control to execution, and those that produced short plans. An increase in the instability of the environment had more of an effect on their performance than it did on strategies that moved to execution before finding complete solutions. The results described here for Tower of Hanoi are representative of the systems performance across all of the implemented domains, with similar strategy interactions appearing across them.

Open-loop execution of complete plans struggled to solve problems when presented with any unexpected events and, predictably, performed the worst of all the strategies. The closed-loop variant benefited from the ability to replan when it encountered problems, but finding full plans often expended enough cycles to prevent the system from solving problems when re-planning was required. The remaining interleaving strategies coped with the occurrence of events through the ability to re-plan, while not wasting as much of their time planning actions that were to be clobbered by unexpected events. Although these results are not surprising, our results demonstrate both the benefit of having a flexible execution system for the purpose of investigation, as well as the importance of having this flexibility to operate in in a variety of domains with different characteristics.

6. Related Research

In this section, we review previous work in the area, highlighting studies that are related to our own efforts. There has been remarkably little attention to variations in execution. Langley, Iba, and Shrager (1994) analyzed the continuum of execution strategies from reactive to automatic control and concluded there is a trade-off between the cost of sensing and the cost of errors. This leads strategies to perform differently in differing environments. For example, automatic control will often outperform reactive control in domains that have a high cost of sensing and a low probability of error. Our work builds on their idea, but focuses on discrete strategies rather than a continuum.

A more substantial body of research deals with interleaving planning and execution. But despite considerable variety in these strategies, the great majority of systems adopt a single approach. At one extreme is the combination of the STRIPS planner (Fikes, Hart, & Nilsson, 1972) and the PLANEX execution system (Nilsson, 1984), which together controlled the early SHAKEY robot. The combined system monitored changes in the environment and had limited ability to recover from unexpected events, but, if it could not recover, returned control to STRIPS to produce a new plan. Thus, this embodied a strategy similar to our implementation of open-loop execution of a complete plan.

A later system was IPEM (Ambros-Ingerson & Steel, 1988), which took a repair-based approach to interleaving planning and execution, treating both execution failures and unexpected events as flaws to to be remedied. However, it did not attempt to fix execution errors until it had corrected all of those that related to planning. Therefore, it generated a complete plan, executed as much of it as possible, recovered from unanticipated changes it could handle, and fell back on the planner to handle those it could not. This approach corresponds to the second strategy that we have implemented with the integrated system: the closed-loop execution of a complete plan with recovery.

In even more recent work, the ICARUS cognitive architecture (Langley, Choi, & Rogers, 2009) took another approach to interleaving planning with execution in physical environments. The framework used means-ends analysis to decompose problems into subproblems and, as soon as it had solved a subproblem, it sent the associated intentions to an execution module. ICARUS could also fall back on planning if execution encountered difficulty. The interleaving strategy that this system employed is similar to FPE/FPS's execution of subplans as soon as the problem solver has completed them.

Naturally, interleaving planning and execution is crucial for almost all game-playing systems. These programs generally execute just one action at a time before returning to planning. However, they may plan many steps ahead before carrying out their single step. The approach that these game-playing systems utilize is therefore similar in spirit to our fourth strategy, repeatedly carrying out N-step forward search followed by a single execution step.

Perhaps the most relevant system in this area is Soar (Laird et al., 2012). This architecture organizes behavior as search through a problem space, on each cycle using knowledge to add elements to working memory that help it select operators to carry out. Laird and Rosenbloom (1990) report a version of Soar that senses an external environment, carries out physical actions, and interleaves planning with execution. There is little question that Soar can support the entire range of behaviors that our system can handle, but it makes no architecture-level commitments about how to do so. Thus, it exhibits the same or even greater flexibility, but it makes weaker theoretical statements about interleaving planning with execution.

7. Concluding Remarks

We began this paper by discussing the behavioral abilities that humans exhibit and that we wish to account for — namely, that they can utilize different techniques to execute complex plans as well as to move from planning to execution and vice versa. In response, we presented five theoretical claims that, together, explain these phenomena. Following this, we introduced FPE/FPS, a integrated system for flexible problem solving and execution that incorporates our postulates. This combined system required several extensions to the original FPS: implementing a five-stage module capable of executing FPS's plans in a simulated environment; adding strategic knowledge to produce varying behavior at three of those stages; and creating stopping criteria for both cycles that govern the transfer of control from the planning module to execution module and back again. After describing these extensions, we explained how our system supports a number of interleaving strategies, five of which we tested on our collection of domains. Our discussion in this section focussed on the interactions between these techniques and domain characteristics, such as the reliability of the agent. We concluded by reviewing previous work that is related to our efforts.

Although our work on flexible execution is promising, there are a number of directions in which we can extend the system. First, we should run additional experiments that take execution time into consideration — for instance, actions in Logistics take much longer to enact than those in the Blocks World — and this is a domain characteristic that warrants further study. Next, we should extend FPE/FPS so that it supports the generation of multiple plans, as well as time constraints for both planning and execution. These features could substantially affect the system's accuracy and speed. A final extension for FPS might include support for plan repair. This would let the planner adapt or extend a failed plan to the new context and would offer an alternative to simple replanning, making the interleaving process more tractable. Finally, in response to Anderson (1998), who notes that some actions, once executed, can render a problem unsolvable, we should include heuristics that reflect the degree to which an operator is reversible. Together, these extensions should produce a more comprehensive account of interleaving planning with execution.

Acknowledgements

This research was supported in part by Grant N00014-10-1-0487 from the Office of Naval Research. We thank Chris MacLellan and Miranda Emery for their efforts on previous versions of the system.

References

Ambros-Ingerson, J. A., & Steel, S. (1988). Integrating planning, execution and monitoring. *Proceedings of the Seventh National Conference on Artificial Intelligence* (pp. 21–26). Saint Paul, MN: AAAI Press.

Anderson, S. D. (1998). Issues in interleaved planning and execution. *Planning, Scheduling and Execution in Dynamic and Uncertain Environments, AAAI Technical Report WS-98-02*. (pp. 62–66). Madison, WI: AAAI Press.

Nilsson, N. J. (1984). *Shakey the robot* (Technical Report). SRI International, Menlo Park, CA.

Fikes, R. E., Hart, P. E., & Nilsson, N. J. (1972). Learning and executing generalized robot plans. *Artificial intelligence*, *3*, 251–288.

Fikes, R. E., & Nilsson, N. J. (1972). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, *2*, 189–208.

Haigh, K. Z., & Veloso, M. M. (1998). Interleaving planning and robot execution for asynchronous user requests. *Autonomous Robots*, *5*, 79–95.

Krebsbach, K., Olawsky, D., & Gini, M. (1992). An empirical study of sensing and defaulting in planning. *Proceedings of the First Conference of AI Planning Systems* (pp. 136–144). Burlington, MA: Morgan Kaufmann.

Laird, J. (2012). *The Soar cognitive architecture*. Cambridge, MA: MIT Press.

Laird, J. E., & Rosenbloom, P. S. (1990). Integrating execution, planning, learning in Soar for external environments. *Proceedings of the National Conference of Artificial Intelligence* (pp. 1022–1029).

Langley, P. Choi, D., & Rogers, S. (2009). Acquisition of hierarchical reactive skills in a unified cognitive architecture. *Cognitive Systems Research*, *10*, 316–332.

Langley, P., Emery, M., Barley, M., & MacLellan, C. (2013). An architecture for flexible problem solving. *Poster Collection: The Second Annual Conference on Advances in Cognitive Systems* (pp. 93–110). Baltimore, MD.

Langley, P., Iba, W., & Shrager, J. (1994). Reactive and automatic behavior in plan execution. *Proceedings of the Second International Conference on AI Planning Systems* (pp. 299–304). Chicago, IL: AAAI Press.

Nilsson, N. J. (1984). *Shakey the robot* (Technical Report). SRI International, Menlo Park, CA.

Penberthy, J. S., & Weld, D. S. (1992). UCPOP: A sound, complete, partial order planner for ADL. *Proceedings of the Third International Conference on Knowledge Representation and Reasoning* (pp. 103–114). Cambridge, MA.

Sapena, O., & Onaindia, E. (2003). An architecture to integrate planning and execution in dynamic environments. *Proceedings of the 22nd Workshop of the UK Planning and Scheduling Special Interest Group* (pp. 184–193). Glasgow, Scotland.

Washington, R. (1995). Incremental planning for truly integrated planning and reaction. *Proceedings of the Fifth Scandinavian Conference on Artificial Intelligence* (pp. 305–316). Trondheim, Norway: IOS Press.