Probabilistic Planning with Information Gathering and Contingent Execution

Denise Draper, Steve Hanks, Dan Weld

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Technical Report 93–12–04
December 19, 1993

Probabilistic Planning with Information Gathering and Contingent Execution*

Denise Draper Steve Hanks Daniel Weld
Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, WA 98195
{*ddraper, hanks, weld*}*@cs.washington.edu*

January 20, 1994

Technical Report 93-12-04

Abstract

One way of coping with uncertainty in the planning process is to plan to gather information about the world at execution time, building a plan contingent on that information. Literature on decision making discusses the concept of information-producing actions, the value of information, and plans contingent on information learned from tests, but these concepts are missing from AI representations and algorithms for plan generation.

This paper presents a planning algorithm that models information-producing (sensing) actions and constructs plans whose executions depend on the nature of the information gathered from sensors. We build on the BURIDAN probabilistic planning algorithm [Kushmerick *et al.*, To appear], extending the action representation to model the behavior of imperfect sensors, and combine it with a framework for contingent action that extends the CNLP algorithm [Peot and Smith, 1992] for conditional execution. The result, C-BURIDAN, is an implemented planner that extends the functionality of both BURIDAN and CNLP.

*This research was funded in part by NASA GSRP Fellowship NGT-50822, National Science Foundation Grants IRI-9206733 and IRI-8957302, and Office of Naval Research Grant 90-J-1904.

Contents

1 Introduction

One way of coping with uncertainty in the planning process is to plan to gather information about the world at execution time, building a plan contingent on the results of that information: before driving a mountain pass in the winter we might listen to a radio broadcast, planning an alternate route if the weather report predicts snow. Before buying a car we might ask a mechanic to examine the car, buying it if and only if the report says it's in good working order.

Information-producing actions and contingent plans are complementary: it doesn't make sense to improve one's information about the world if that information can't be exploited later in the plan, and it doesn't make sense to build a contingent plan if there is no way to improve one's state of information about the world at execution time.

Planning with information-producing actions and contingencies requires several extensions to classical plan and action representations:

- we need to represent the agent's incomplete state of information about the world
- we need to represent the difference between changes an action makes to the world and changes an action makes to the agent's state of information about the world
- we need to represent a plan whose actions depend on information obtained during execution.

This paper presents a representation and algorithm for probabilistic planning with information-producing actions and contingent execution. We extend the BURIDAN [Kushmerick *et al.*, To appear] probabilistic action representation to allow actions with both informational and causal effects, and combine it with a framework for building contingent plans that builds on the CNLP algorithm [Peot and Smith, 1992]. The resulting algorithm, C-BURIDAN, takes as input a probability distribution over initial world states, a goal expression, a set of action descriptions, and a probability threshold, and produces a contingent plan that makes the goal expression true with a probability no less than the threshold.

Consider the following example, which the paper will develop fully. A manufacturing robot is given the task of processing a widget. Its goal is to have the widget *painted* (PA) and *processed* (PR). Processing the widget consists of identifying it as either *flawed* (FL) or *not flawed* ($\overline{\text{FL}}$), then rejecting or shipping the widget (reject or ship), respectively. The robot also has an operator paint that usually makes PA true.

Although the robot cannot immediately tell whether or not the widget is flawed, it does have a sensing operator inspect that tells it whether the widget is *blemished* (BL). The sensor usually reports **bad** if the widget is blemished, and **ok** if not. Initially any widget that is flawed is also blemished. Two things complicate the sensing process:

- Painting the widget removes a blemish but not a flaw, so executing inspect after the widget has been painted conveys no information about a flaw.
- The sensor is sometimes wrong: if the widget is blemished then 90% of the time the sensor will report **bad**, but 10% of the time it will erroneously report **ok**. If the widget is not blemished, however, the sensor will always report **ok**.

Assume that initially there is a 0.3 chance that the widget is both flawed and blemished (i.e. that FL and BL are both true) and a 0.7 chance that it is neither flawed nor blemished.

A planner that cannot observe the widget and exploit the information from the sensor can at best build a plan with a success probability 0.7: it assumes the widget will not be flawed, paints it and ships it.

An information-gathering planner can generate a plan that works with probability .97: it first senses the widget, then paints it. Then if the sensor reported **ok**, it ships the widget, otherwise it rejects the widget. This plan fails only in the case that the widget was initially flawed but the sensor erroneously reports **ok**, which occurs with probability $(0.3)(0.1) = 0.03$.[1]

Building this plan requires reasoning about several new concepts:

- Information-producing actions: the planner must distinguish between an operator that *makes* the widget blemished (or removes a blemish) and one that *observes* whether it is blemished. Both change the probability that BL is true, but the latter also provides information about FL while the former does not.

- Imperfect sensing actions: the fact that the sensor can incorrectly report that the widget is unblemished affects the probability that the plan will succeed.

- Informational dependencies: inspecting the widget *before* it is painted conveys information about whether or not the widget is flawed; doing so after the widget is painted does not.

- Contingent execution: the ship and reject actions are both in the plan, but should be executed under different circumstances—the former when the inspect action reports **ok**, the latter when it reports **bad**.

The C-BURIDAN planner confronts these problems, and generates the contingent plan described above. C-BURIDAN's main technological advances are:

1. An action represent that extends the BURIDAN planner to include information-producing (sensory) actions. C-BURIDAN uses the traditional Bayesian framework for representing imperfect evidence sources (sensors), but the symbolic component of its representation allows actions to be manipulated by a plan-generation algorithm as well. C-BURIDAN's action representation allows arbitrary symbolic and probabilistic dependencies between the world's state at execution time and the report returned by the sensor

2. A plan representation and generation algorithm that supports contingent planning. Step execution can depend on reports generated by prior information-producing actions. The C-BURIDAN plan representation generalizes traditional conditional plan representations ([Warren, 1976], [Peot and Smith, 1992], [Etzioni *et al.*, 1992]) in that it allows conditional branches to be "merged" (see Section 3.6).

This paper describes the implemented C-BURIDAN system by developing the widget-shipping example introduced earlier in this section. It begins with the action and plan representation, then describes the planning algorithm, and ends with a discussion of efficiency issues and directions for future work.

[1] Actually a planner can generate an even better plan by sensing the part repeatedly.

2 Representation: States, Actions, and Plans

Here we present the formal definition of a state, an action, a plan, a planning problem, and what it means for a plan to solve a planning problem. Much of this representation is identical to the BURIDAN planner, and the reader is referred to [Kushmerick *et al.*, To appear] for more detail.

2.1 Propositions

We begin by defining a set of *domain propositions*, each of which describes a particular aspect of the world. Domain propositions for our example are:

- FL—the widget is flawed
- BL—the widget is blemished
- PR—the widget is processed
- PA—the widget is painted
- ER—an execution error occurs

A domain proposition means that aspect of the world is true and a negated domain proposition, e.g. $\overline{\mathsf{FL}}$, means that aspect of the world is false.

2.2 States

A *state* is a complete description of the agent's model of the world at a particular point in time. We represent a state as a set of domain propositions in which every proposition appears exactly once, negated or non-negated. In our example we know that initially the widget has not been processed or painted, and that there is as yet no error. But there is some chance it is flawed and blemished and some chance it is neither. Thus there are two possible initial states:

$$s_1 = \{\, \mathsf{FL},\ \mathsf{BL},\ \overline{\mathsf{PR}},\ \overline{\mathsf{PA}},\ \overline{\mathsf{ER}} \,\} \tag{1}$$

$$s_2 = \{\, \overline{\mathsf{FL}},\ \overline{\mathsf{BL}},\ \overline{\mathsf{PR}},\ \overline{\mathsf{PA}},\ \overline{\mathsf{ER}} \,\} \tag{2}$$

We will use $\tilde{\mathsf{s}}$ to refer to a random variable over states, and $\tilde{\mathsf{s}}_I$ the particular distribution over initial states. This random variable is defined as follows for our example:

$$\begin{aligned} \mathrm{P}[\tilde{\mathsf{s}}_I = s_1] &= 0.3 \\ \mathrm{P}[\tilde{\mathsf{s}}_I = s_2] &= 0.7 \end{aligned}$$

The probability of an arbitrary event X given the random variable $\tilde{\mathsf{s}}$ is computed from the probabilities of X given individual states in the standard manner:

$$\mathrm{P}[\mathsf{X} \,|\, \tilde{\mathsf{s}}] \equiv \sum_s \mathrm{P}[\mathsf{X} \,|\, s] \times \mathrm{P}[\tilde{\mathsf{s}} = s], \tag{3}$$

For completeness, we define the probability of a set $\mathcal{X}$ of domain propositions given a state to be 1 iff the conjunction of propositions in $\mathcal{X}$ is true in that state:

$$\mathrm{P}[\mathcal{X} \,|\, s] = \begin{cases} 1 & \text{if } \mathcal{X} \subseteq s \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Then we can compute, for example:

$$\begin{aligned} \mathrm{P}\left[\{\overline{\mathsf{PR}}, \overline{\mathsf{PA}}\} \,|\, \tilde{\mathsf{s}}_I\right] &= \mathrm{P}\left[\{\mathsf{FL}, \overline{\mathsf{PA}}\} \,|\, s_1\right] \times \mathrm{P}[\tilde{\mathsf{s}}_I = s_1] + \mathrm{P}\left[\{\mathsf{FL}, \overline{\mathsf{PA}}\} \,|\, s_2\right] \times \mathrm{P}[\tilde{\mathsf{s}}_I = s_2] \\ &= 1 \times 0.7 + 0 \times 0.3 = 0.7 \end{aligned}$$

2.3 Actions

An action describes the effects the plan operator has on the world when it is executed. The nature of the effects can depend both on the state in which the step is executed as well as random factors (not modeled in the state). Figure 1 shows a diagram of the paint action: an error results if an attempt to paint the widget is made when it has already been processed; otherwise with probability 0.95 the widget will become painted and all blemishes removed and with probability 0.05 the action will not change the state of the world at all. Notice that the leaves of the tree do not contain states; rather they indicate *changes* to a state (like STRIPS adds and deletes).

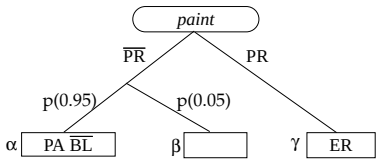

Figure 1: The *paint* action usually paints the widget (PA) and removes blemishes ($\overline{\mathsf{BL}}$).

We can describe an action formally as a set of *consequences*. Each consequence is a triple of the form $\langle \mathcal{T}_\iota, \rho_\iota, \mathcal{E}_\iota \rangle$, where $\mathcal{T}_i$ is a set of domain propositions known as the consequence's *trigger*, p_i is a probability, and $\mathcal{E}_i$ is a set of *effects* associated with the consequence. The representation for the paint action is

$$\begin{aligned} \mathsf{paint} = \{\ & (\{\mathsf{PR}\},\ 1.0,\ \ \{\mathsf{ER}\}) \\ & (\{\overline{\mathsf{PR}}\},\ 0.95,\ \{\mathsf{PA},\ \mathsf{BL}\}) \\ & (\{\overline{\mathsf{PR}}\},\ 0.05,\ \{\}) \ \} \end{aligned}$$

The changes resulting from a set of effects $\mathcal{E}$ is defined by a function RESULT$(\mathcal{E}, s)$ in the manner of a STRIPS add and delete list: negate all the propositions that appear negated in $\mathcal{E}$ then remove negations from all propositions that appear in $\mathcal{E}$ without negation (see [Kushmerick *et al.*, To appear] for the full definition).

An action induces a change from a state s to a probability distribution over states, which we define in terms of the probabilities that its consequences will occur. First of all, for any consequence $\langle \mathcal{T}_\iota, \rho_\iota, \mathcal{E}_\iota \rangle$ we define

$$\mathrm{P}[\langle \mathcal{T}_\iota, \rho_\iota, \mathcal{E}_\iota \rangle \,|\, s] = \mathrm{P}[\mathcal{T}_i \,|\, s] \times p_i \tag{5}$$

Notice that $\mathcal{T}_i$ is a set of domain propositions, therefore $\mathrm{P}[\mathcal{T}_i \,|\, s]$ will be either 0 or 1 for any state s, and so this probability will either be 0 or p_i for any consequence. Furthermore, we require the triggers of an action to have following two properties:

- an action's triggers $\mathcal{T}$ are mutually exclusive and exhaustive, so for any state s one trigger will have probability 1 and the rest will have probability 0,
- for any single trigger $\mathcal{T}_i$, the associated probabilities p_j will sum to 1.

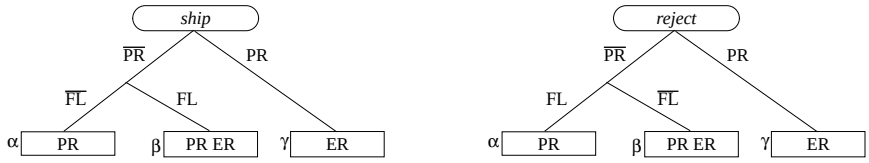

Figure 2: The ship and reject actions

As a result we know that $\sum_{\langle \mathcal{T}_\iota, \rho_\iota, \mathcal{E}_\iota \rangle \in \mathsf{A}} \mathrm{P}[\langle \mathcal{T}_\iota, \rho_\iota, \mathcal{E}_\iota \rangle \,|\, s] = 1$ for any action A and any state s. The tree-structured form of our actions (e.g. Figure 1) makes it easy to construct actions which have these properties.

Now we can define the probability that executing an action A starting from state s results in a new state t:

$$\mathrm{P}[t \,|\, s, \mathsf{A}] = \begin{cases} \mathrm{P}[\langle \mathcal{T}_\iota, \rho_\iota, \mathcal{E}_\iota \rangle \,|\, s] & \text{if } \langle \mathcal{T}_\iota, \rho_\iota, \mathcal{E}_\iota \rangle \in \mathsf{A} \text{ and } t = \textsc{result}(\mathcal{E}_i, s) \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

The form of an action's consequences noted above guarantees that $\sum_t \mathrm{P}[t \,|\, s, \mathsf{A}] = 1$ for any action A and any state s.

In the example scenario the distribution after executing paint looks like this:

$$\begin{aligned} &\mathrm{P}\big[\{\mathsf{FL}, \overline{\mathsf{BL}}, \overline{\mathsf{PR}}, \mathsf{PA}, \overline{\mathsf{ER}}\} \,|\, \tilde{\mathsf{s}}_I,\ \mathsf{paint}\big] = 0.285 \\ &\mathrm{P}\big[\{\mathsf{FL}, \mathsf{BL}, \overline{\mathsf{PR}}, \overline{\mathsf{PA}}, \overline{\mathsf{ER}}\} \,|\, \tilde{\mathsf{s}}_I,\ \mathsf{paint}\big] = 0.015 \\ &\mathrm{P}\big[\{\overline{\mathsf{FL}}, \overline{\mathsf{BL}}, \overline{\mathsf{PR}}, \mathsf{PA}, \overline{\mathsf{ER}}\} \,|\, \tilde{\mathsf{s}}_I,\ \mathsf{paint}\big] = 0.665 \\ &\mathrm{P}\big[\{\overline{\mathsf{FL}}, \overline{\mathsf{BL}}, \overline{\mathsf{PR}}, \overline{\mathsf{PA}}, \overline{\mathsf{ER}}\} \,|\, \tilde{\mathsf{s}}_I,\ \mathsf{paint}\big] = 0.035 \end{aligned}$$

And given this probability distribution we can compute the marginal probabilities of various combinations of domain propositions. For example:

$$\begin{aligned} &\mathrm{P}[\{\mathsf{PA}\} \,|\, \tilde{\mathsf{s}}_I,\ \mathsf{paint}] = 0.95 \\ &\mathrm{P}[\{\mathsf{BL}\} \,|\, \tilde{\mathsf{s}}_I,\ \mathsf{paint}] = 0.015 \\ &\mathrm{P}[\{\mathsf{PR}, \mathsf{ER}\} \,|\, \tilde{\mathsf{s}}_I,\ \mathsf{paint}] = 0.0 \\ &\mathrm{P}[\{\mathsf{FL}\} \,|\, \tilde{\mathsf{s}}_I,\ \mathsf{paint}] = 0.3 \end{aligned}$$

Figure 2 shows two more actions pertaining to the example: ship and reject. The ship successfully processes the widget if it is not flawed and not already processed. Shipping a flawed widget or trying to ship a widget that has already been processed will cause an execution error. The reject step works similarly: the step processes the widget successfully if it *is* flawed and it has not already been processed.

2.3.1 Action sequences

We will often reason about executing a sequence of actions—we will use $\langle \mathsf{A}_1, \mathsf{A}_2, \ldots, \mathsf{A}_n \rangle$ to mean executing A_1, then A_2, and so on, and $\langle\rangle$ to mean a sequence of 0 actions.

State probabilities after executing a sequence of actions are defined as an extension of Equation 6:

$$\mathrm{P}[u \,|\, s, \langle\rangle] \;=\; \begin{cases} 1 & \text{if } u = s \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

$$\mathrm{P}[u \,|\, s, \langle \mathsf{A}_1, \mathsf{A}_2, \ldots, \mathsf{A}_n \rangle] \;=\; \sum_t \mathrm{P}[t \,|\, s, \mathsf{A}_1] \times \mathrm{P}[u \,|\, t, \langle \mathsf{A}_2, \ldots, \mathsf{A}_n \rangle] \tag{8}$$

2.4 Information-producing actions

We have so far represented actions as a mapping from a state to a probability distribution over states, where the new distribution is defined by the effects the action will have on the world if it is executed in the input state. By contrast consider an action inspect that reports on whether or not the widget is blemished (whether or not BL is true). We want to retain the definition of actions as mappings from mutually exclusive triggers into sets of effects, but we also want to distinguish between the effects the action has on the world and the effects it has on the agent's state of information [Etzioni *et al.*, 1992]. Executing inspect does not change the probability that BL is true, but it does provide the agent with information about whether BL is true or not. Executing inspect and receiving a report that BL is true *does* change BL's posterior probability, but the nature of that report will not be known until the step is actually executed.

We model the information produced by an action by allowing it to report on which of its consequences actually occurred at execution time. We do so by placing an *observation label* on each of an action's consequences. The observation label corresponds to the sensor's report: when the action is executed the agent will be informed of the observation label of the consequence that was actually realized in the world.

There is an important distinction between our representation and the approach employed by other sensing representations (*e.g.* [Peot and Smith, 1992]), which assert the results of a information-gathering act just as if the information-gathering act had *made* the result true. Confusing the changes an action makes to the world with the changes an action makes to the agent's *information* about the world can lead to anomalous behaviors, like attempting to make a proposition true by repeatedly sensing it. C-BURIDAN avoids this and similar difficulties, since inserting an information-producing action into a plan does does not change the probability that the observed proposition is true when the the plan is executed. However the *conditional* probability that the observed proposition is true given that a particular report has been received does change, as we will explain below.

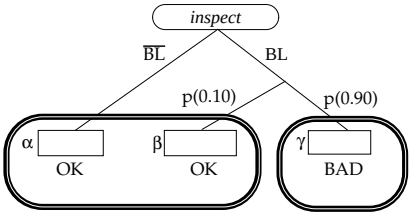

Figure 3: The inspect action models an imperfect sensor

Consider the inspect action in Figure 3.[2] There are two different observation labels, **bad**, and **ok**. Also notice that two consequences are labeled with **ok**, so the agent would be unable to ascertain which of these two consequences actually occurred.

[2]This version of inspect is overly simple in that it allows the widget to be inspected after it has been processed. A more realistic version of the action would have a fourth consequence with trigger PR, which would make ER true.

The observation labels must be included in an action's definition, so now an action's consequence consists of four parts: $\langle \mathcal{T}_i, \rho_i, \mathcal{E}_i, o_i \rangle$, where $\mathcal{T}_i$ is the trigger, p_i is the probability the consequence will be realized if the trigger is true, $\mathcal{E}_i$ is the effects the consequence will have if it is realized, and o_i is the observation label reported if the consequence is realized.

Information-producing actions are those with more than one observation label; the information they convey is defined by their *discernible equivalence classes*, or DECs, indicated by the ovals in Figure 3. Actions like paint have no informational effect—they have a single observation label attached to all their consequences, thus they have a single DEC. Executing the action provides no information as to which consequence actually came about.

Notice the difference between the causal and informational effects of executing an action. The changes an action makes to the world are recorded in the change sets of its consequences, and the planner uses the contents of the actions' effect sets when adding steps and links to a plan. Although the actions in our example problem are either essentially causal in their effects (paint) or informational in their effects (inspect), there is no reason that an action cannot both change the world and provide information about it as well—an action can have both propositions in its change sets and more than one observation label, and the planner can exploit both of these properties.

The immediate information provided by the execution of inspect can be characterized as follows: if it generates the report **bad** then consequence γ definitely occurred and BL is definitely true, but if it generates the report **ok** then either α or β occurred, and BL may or may not be true.

We can characterize the inspect action more precisely using these conditional probabilities that appear in the action's consequences:

$$\begin{array}{llll} \mathrm{P}[\mathbf{bad} \mid \mathsf{BL}] & = 0.9 & \mathrm{P}[\mathbf{ok} \mid \mathsf{BL}] & = 0.1 \\ \mathrm{P}[\mathbf{bad} \mid \overline{\mathsf{BL}}] & = 0.0 & \mathrm{P}[\mathbf{ok} \mid \overline{\mathsf{BL}}] & = 1.0 \end{array}$$

which is a standard probabilistic representation of an uncertain evidence source (see, e.g., [Pearl, 1988, Chapter 2]).

The probabilities of domain propositions conditioned on sensor reports is handled using Bayes rule in the standard way. Suppose inspect is executed in the initial state (where PR is known to be false and $\mathrm{P}[\mathsf{BL}] = 0.3$), and the report **ok** is received:

$$\mathrm{P}[\mathsf{BL} \mid \mathbf{ok}] = \frac{\mathrm{P}[\mathbf{ok} \mid \mathsf{BL}]\mathrm{P}[\mathsf{BL}]}{\mathrm{P}[\mathbf{ok} \mid \mathsf{BL}]\mathrm{P}[\mathsf{BL}] + \mathrm{P}[\mathbf{ok} \mid \overline{\mathsf{BL}}]\,\mathrm{P}[\overline{\mathsf{BL}}]} = \frac{(0.1)(0.3)}{(0.1)(0.3) + (1.0)(0.7)} = .041$$

If instead the report **bad**is received:

$$\mathrm{P}[\mathsf{BL} \mid \mathbf{bad}] = \frac{\mathrm{P}[\mathbf{bad} \mid \mathsf{BL}]\mathrm{P}[\mathsf{BL}]}{\mathrm{P}[\mathbf{bad} \mid \mathsf{BL}]\mathrm{P}[\mathsf{BL}] + \mathrm{P}[\mathbf{bad} \mid \overline{\mathsf{BL}}]\,\mathrm{P}[\overline{\mathsf{BL}}]} = \frac{(0.9)(0.3)}{(0.9)(0.3) + (0.0)(0.7)} = 1.0$$

The inspect action can also provide information about propositions other than BL. Since BL and FL are initially perfectly correlated in the example, we have $\mathrm{P}[\mathsf{FL} \mid \mathsf{BL}] = 1$ and therefore can conclude $\mathrm{P}[\mathsf{FL} \mid \mathbf{bad}] = 1$ and $\mathrm{P}[\mathsf{FL} \mid \mathbf{ok}] = 0.041$. Executing paint destroys this correlation, however, so executing inspect after paint would not provide any additional information about FL(but it still would about BL). The point is that the information content of an action cannot be fully characterized by examining the operator alone—it depends on what probabilistic relationships hold in the plan at the time the action is executed.

Finally we should note that sensing actions like inspect are useless by themselves: the fact that the probability of FL changes depending on whether inspect reports **ok** or **bad** is of no use to

the planner if it can't make the execution of subsequent plan steps sensitive to which of the two reports are received. Section 3.4.1 discusses how the execution of steps in a plan can depend on the observation labels generated by previous steps.

2.5 Plan steps & contexts

A plan step is a triple: $\langle action, index, context \rangle$.

The step index allows multiple instances of the same action to appear in the plan, in particular allowing their observation labels to be distinguished. Suppose that there were two instances of inspect in a plan, one with index i, the other with index j. The two observation labels of the first step would be "OK-i" and "BAD-i" while the second step's observation labels would be "OK-j" and "BAD-j." Our example plans will have only one instance of any action, so we will omit these indexes in the paper.

A step's context dictates the circumstances under which the step should be executed. In particular, each context is a set (implicit conjunction) of observation labels from previous steps in the plan; only when the step's context matches the observations actually produced during execution can it can be executed.

Suppose, for example, we have the following sequence of steps:

$\langle \mathsf{inspect}, 1, T \rangle$,
$\langle \mathsf{ship}, 2, \{\mathbf{ok}\} \rangle$
$\langle \mathsf{reject}, 3, \{\mathbf{bad}\} \rangle$

and suppose the agent executes the inspect step, receiving the report **bad**. It then considers executing the next step in the sequence. It skips ship, since that step's context does not match the report produced by execution of inspect. The agent next decides to execute the third step, reject, since the step's context *does* match the report produced by inspect.

In summary, we require that an agent keep track of the *execution context*—the reports (observation labels) produced by the steps executed in the plan so far. Plan steps are executed only when the execution context matches the step's context.

2.6 Assessing goal probability

Here we define the probability that the a sequence of steps satisfies some goal expression $\mathcal{G}$. The definition is an extension of Equation 8 that takes into account each step's context and its relation to previously executed steps in the sequence.

In particular, the effect of executing a step given an execution context is either (i) the effect of executing the corresponding action (if the contexts match) otherwise, (ii) no change (if the step's context doesn't match the execution context). To make this precise, we start with the analogue of Equation 7, stating that if there are no steps in the sequence then the probability depends only on the initial probability. Here C is an execution context (a conjunction of observation labels), and the probability computed is that the world will be in state u after a particular sequence of steps is executed, given that the execution context is C and that the world is currently in state s.

$$\mathrm{P}[u \,|\, \mathsf{C}, s, \langle\rangle] = \begin{cases} 1 & \text{if } u = s \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

Next, if the first step in a sequence is not executable it does not change any probabilities, nor does it change the execution context. Recall that both the step's context and the execution context are boolean combinations of observation labels, so we can define "executability" in terms of logical entailment: S is executable in context C just in case $\mathsf{C} \vdash \text{context}(\mathsf{S})$.

$$\begin{aligned} \mathrm{P}[u \,|\, \mathsf{C}, s, \langle \mathsf{S}_1, \mathsf{S}_2, \ldots, \mathsf{S}_n \rangle] \quad &= \quad \mathrm{P}[u \,|\, \mathsf{C}, s, \langle \mathsf{S}_2, \ldots, \mathsf{S}_n \rangle] \\ &\text{if } \mathsf{C} \nvdash \text{context}(\mathsf{S}_1) \end{aligned} \tag{10}$$

Finally, if a step *is* executable in the current context it changes both the probability distribution over states and the execution context, both according to its set of consequences $\text{action}(\mathsf{S}) = \{\langle \mathcal{T}_i, \rho_i, \mathcal{E}_i, o_i \rangle\}$:

$$\begin{aligned} &\mathrm{P}[u \,|\, \mathsf{C}, s, \langle \mathsf{S}_1, \mathsf{S}_2, \ldots, \mathsf{S}_n \rangle] = \\ &\sum_{\langle \mathcal{T}_i, \rho_i, \mathcal{E}_i, o_i \rangle \in \text{action}(\mathsf{S}_1)} \mathrm{P}[\langle \mathcal{T}_i, \rho_i, \mathcal{E}_i, o_i \rangle \,|\, s] \mathrm{P}[u \,|\, (\mathsf{C} \wedge o_i), \textsc{result}(\mathcal{E}_i, s), \langle \mathsf{S}_2, \ldots, \mathsf{S}_n \rangle] \\ &\qquad\qquad \text{if } \mathsf{C} \vdash \text{context}(\mathsf{S}_1) \end{aligned} \tag{11}$$

2.7 Planning problems and solutions

A *planning problem* consists of:

- A probability distribution over initial states $\tilde{\mathsf{s}}_I$.
- A *goal expression* $\mathcal{G}$—a set (conjunction) of domain propositions describing the desired final state of the system.
- A set of *actions* like paint, inspect, ship, and inspect, defining the agent's capabilities.
- A *probability threshold* τ specifying a lower bound on the success probability for an acceptable plan.

The planning algorithm produces a sequence of *steps* $\langle \mathsf{S}_1, \ldots, \mathsf{S}_n \rangle$, where each step is an instance of an action along with the circumstances under which the action should be executed. Such a sequence is a *solution* to the problem if the probability of the goal expression after executing the steps is at least equal to the threshold. In other words, $\langle \mathsf{S}_1, \ldots, \mathsf{S}_n \rangle$ solves the planning problem just in case $\mathrm{P}[\mathcal{G} \,|\, \tilde{\mathsf{s}}_I, \langle \mathsf{S}_1, \ldots, \mathsf{S}_n \rangle] \geq \tau$. Section 2.6 defines this probability.

3 Plans and planning

Our planner takes a problem (initial probability distribution, goal expression, threshold, set of actions) as input and produces a solution sequence—a sequence of steps whose probability of achieving the goal exceeds the threshold. Here we describe its data structures and the algorithm it uses to produce a solution.

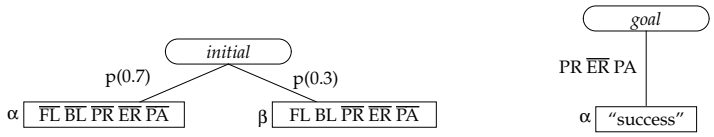

Figure 4: The initial plan

3.1 Initial and goal steps

The planner initially converts the problem's initial and goal states into two steps, initial and goal. The initial step codes the initial probability distribution, and the goal step has a single consequence with the goal state as its trigger. Figure 4 shows initial and goal actions for the example.

3.2 Plans

Following BURIDAN [Kushmerick *et al.*, To appear], the planner manipulates a data structure called a *plan*, consisting of a set of steps, ordering constraints over the steps, and a set of *causal links*. The initial and goal actions each appear exactly once in every plan, with the initial step ordered before all others and the goal step ordered after all others. A plan with *only* these two steps and this single ordering is called the initial, or null plan, and is the algorithm's starting point.

3.2.1 Causal links

Causal links record decisions about the role the plan's steps play in achieving the goal. A link connects a consequence of a "producing" step with a proposition p in a trigger of a "consuming" step. A causal link makes explicit the dependency between the two steps, recording the fact that increasing the probability of the producing step's designated consequence will tend to increase the probability that p will be true when the consuming step is executed, which will tend to make the plan more likely to succeed.[3]

For example, the planner might create a link from the α consequence of a paint step to the PA trigger proposition of the goal step, indicating that paint is supposed to make PA true for use by goal. We will use the notation $\mathsf{paint}_{\alpha} \xrightarrow{\mathsf{PA}} \mathsf{goal}$ to refer to this link.

More generally the notation $\mathsf{S}_{i\iota} \xrightarrow{\mathsf{p}} \mathsf{S}_j$ refers to a link whose producer is consequence ι of S_i, which produces proposition p for S_j.[4] Since the goal expression appears in the plan as a set of triggers in the single consequence of the goal step, planning to achieve the goal proceeds by adding steps that produce the goal propositions and linking from the appropriate consequences of these steps to the goal step, then creating links to the triggers of these new consequences as well.

[3] We say that adding links will "tend" to increase the probability of success because the circumstances under which two links will produce the desired result can be correlated, hence there is no guarantee that adding a link will increase the probability of the supported proposition. For example, suppose a link records a desire to make p true, but its producing step will do so only if some other proposition q is true. Now suppose we create a second link to make p true, but it too does so only if q is true. Adding the second link does not increase the probability that p will be true: either both links will generate p or neither will. Adding multiple links to the same consuming proposition can never *decrease* the probability that it will be true, but the possibility of correlation with other links means that it might not increase the probability either.

[4] We will refer to the j^{th} step in a plan either with the notation S_j, or by using its name, e.g. paint. Referring to a step by name is not generally a good idea since there could be several steps with the same name in the plan. However, since our example has only one step for each action, there is no ambiguity.

3.2.2 Subgoals

A plan's links determines a set of *subgoals* for the planner, each a pair of the form $\langle \mathsf{p}, \mathsf{S}_j \rangle$, where p is some proposition and S_j is some step in the plan. The presence of such a pair indicates that raising the probability that p is true at the time S_j is executed will tend to increase the probability that the plan will succeed. A pair $\langle \mathsf{p}, \mathsf{S}_j \rangle$ is a member of a plan's subgoals just in case either

- j is the index of the goal step and p is a member of the problem's goal expression $\mathcal{G}$ (meaning that p is one of the explicit goals of the planning problem), or
- p is a proposition in one of the triggers of an consequence ι of S_i, and the plan contains a link of the form $\mathsf{S}_{i\iota} \xrightarrow{\mathsf{q}} \mathsf{S}_j$ for some S_j and some proposition q.

In other words, the subgoals represent the set of propositions that participate in chains of causal links ending at the goal. We will introduce additional subgoals in Sections 3.3.2 and 3.4.1.

3.2.3 Threats to links

The process of adding steps and links to the plan can generate conflicts. Suppose that a plan contains a link of the form $\mathsf{S}_{i\iota} \xrightarrow{\mathsf{p}} \mathsf{S}_j$. The link actually represents two commitments on the planner's part: (1) to make S_i realize its consequence ι, resulting in p becoming true, and (2) to keep p true from S_i's execution until S_j's execution. Therefore any step that

1. possible occurs after S_i, and
2. possibly occurs before S_j, and
3. has an effect that makes p false

is called a *threat* to the link $\mathsf{S}_{i\iota} \xrightarrow{\mathsf{p}} \mathsf{S}_j$.

3.2.4 Summary

The *plan* data structure consists of a set of steps, a set of ordering constraints, and a set of causal links. Every plan contains the initial and goal steps that describe the planning problem. From a plan's links one can compute a set of *subgoals* and a set of *threats*. The subgoals indicate propositions in the plan for which adding additional links might make the plan more likely to succeed. The threats indicate conflicts in the plan that might prevent success. Planning therefore involves creating links to subgoal propositions and resolving threats to those links. We call these two options the possible *refinements* to the plan.

3.3 The planning algorithm

Here is a brief description of the planning algorithm:

1. Begin with the *null plan*, containing only steps initial and goal, the ordering (initial $<$ goal), and no causal links.
2. Iterate:

(a) *Assess* the current plan: compute the probability that the current plan achieves the goal. Report success if that probability is at least as great as the threshold.

(b) Otherwise nondeterministically choose a *refinement* to the current plan. Report failure if there are no possible refinements. Otherwise apply the refinement to the current plan and repeat.

Any implementation of the algorithm will of course have to manage the nondeterministic choice using backtracking or some other search technique.

3.3.1 Assessment

A plan defines a partial order over its steps, which in turn defines a set of legal execution sequences. Calculating the probability that such a set of steps could achieve a goal is a complex task. Here, we present a very simple algorithm for performing this calculation, based closely on the definition of step execution. The algorithm simply iterates over all step sequences consistent with the plan's ordering constraints. For each totally ordered sequence, the algorithm calculates the probability distribution over the states that could possibly result. When it has finished iterating over the steps in a particular total order, it sums the probabilities of all the states in the distribution in which the goal is true. If the probability is greater than τ, the algorithm returns the successful sequence, otherwise it continues to the next total order. If no sequence has a success probability greater than τ the assessor reports failure.

The probability distribution is represented as a set of triples $(s_j,\ p_j,\ c_j)$ where s_j is a state as defined in Section 2.2, p_j is a probability, and c_j is a context (as defined in the previous section). Context c_j is the conjunction of all the observation labels that were observed in the execution that produced state s_j.
For each totally ordered sequence $\langle \mathsf{S}_1, \ldots, \mathsf{S}_n \rangle$ consistent with the plan:

1. Initialize `fringe` := $\{(s_i,\ p_i,\ \mathsf{T})\}$ where $\mathrm{P}[\tilde{\mathbf{s}}_I = s_i] = p_i$ is the initial probability distribution provided by the user and T is a context that is always true.

2. Loop for $\mathsf{S} = \mathsf{S}_1, \mathsf{S}_2, \ldots, \mathsf{S}_n$:

   (a) Loop for $(s_j,\ p_j,\ c_j) \in$ `fringe`:

      i. When $c_j \vdash \mathrm{context}(\mathsf{S})$ do:

         A. Remove $(s_j,\ p_j,\ c_j)$ from `fringe`

         B. Loop for $(\mathcal{T}_i, p_i, \mathcal{E}_i, o_i) \in \mathrm{action}(\mathsf{S})$:
            Add to `fringe` the triple $(\textsc{result}(\mathcal{E}_i, s_j),\ p_j \times p_i \times \mathrm{P}[\mathcal{T}_i \,|\, s_j],\ c_j \wedge o_i)$

3. If $\tau \leq \sum_{\{(s_j,\ p_j,\ c_j) \in \mathtt{fringe}\}} p_j \times \mathrm{P}[\mathcal{G} \,|\, s_j]$ then return $\langle \mathsf{S}_1, \ldots, \mathsf{S}_n \rangle$.

The algorithm computes the probability distribution over states generated by each action in the sequence, finally summing the probabilities of all final states in which the goal is true.

This simple version of plan assessment is often quite inefficient; we include it here only to keep the presentation simple. [Kushmerick *et al.*, To appear] discusses four different plan-assessment algorithms and compares their performance. One of the most interesting assessment algorithms presented in that paper uses the plan's set of causal links to estimate the success probability without enumerating any total ordered sequences.

3.3.2 Refinement

A plan refinement adds structure to a plan, trying to increase the probability that the plan will achieve its goal expression. Recall that the probability of goal achievement can be increased in one of two ways:

- if $\langle \mathsf{p}, \mathsf{S}_i \rangle$ is a subgoal, then adding a new link from some (possibly new) plan step to this proposition might increase the probability that p is true at S_i, and therefore might increase the success probability,
- if a causal link is currently part of the plan but some other step in the plan threatens the link then eliminating the threat might increase the probability of the link's consumer proposition, and therefore might increase the success probability.

Adding links and steps to the plan is the same as for deterministic causal-link planners: for some pair $\langle \mathsf{p}, \mathsf{S}_j \rangle$ selected to be the link's consumer, a link $\mathsf{S}_{i\iota} \xrightarrow{\mathsf{p}} \mathsf{S}_j$ can be inserted for any existing step S_i that can be ordered before S_j and whose consequence ι asserts p, or between any *new* step with a consequence ι that asserts p, where the new step is constrained to occur before S_j. However, notice that the existence of a link to a trigger proposition does not *ensure* that the proposition will be true—that will depend on whether or not the producing consequence is actually realized. As a result the planner needs to be able to introduce multiple links to a proposition to try to increase the probability that it will be true.

Threat resolution in C-BURIDAN is also similar to threat resolution in deterministic causal-link planners. Promotion and demotion (*i.e.*, the addition of ordering constraints on the threatening steps) works in precisely the same way as in the deterministic case. Two additional threat-resolution mechanisms, *confrontation* and *branching*, have no analogue in classical planning, however.

Confrontation was introduced in the BURIDAN probabilistic planner and is adopted without change from that system [Kushmerick *et al.*, To appear]. The idea behind confrontation is that a plan can be sufficiently likely to work even if some action in the plan makes a goal or subgoal false, as long as the falsifying consequence of that action is sufficiently *unlikely* to occur. Since the consequences of any action are mutually exclusive, the planner can make a particular consequence of an action *less* likely to occur by making a different consequence of that same action *more* likely to occur. If consequence τ of some step S_t threatens a link by making p false, then the threat can be confronted by choosing some consequence of S_t that does *not* make p false and adopting its trigger as a subgoal.

The last way of resolving threats is called resolution by *branching*, and has no analogue in classical planning or in BURIDAN. We explain the details in Section 3.4.1, but provide a brief overview here. Intuitively, branching ensures that the agent will never *execute* the threatening step when the link's consuming step is depending on an effect of the link's producing step. Resolution by branching works by making the *context* in which S_t occurs incompatible with the context in which the link proposition is required by the link's consumer.

In summary, the algorithm's refinement phase consists of nondeterministically choosing to work on a subgoal or to resolve a threat:

1. Choose either a subgoal to support or a threatened link to protect.
2. If the choice is a subgoal of the form $\langle \mathsf{p}, \mathsf{S}_j \rangle$, choose some step S_i whose consequence ι asserts p. S_i must either be a new step (instance of an action), or an existing step in the plan that can be ordered before S_j. Add the link $\mathsf{S}_{i\iota} \xrightarrow{\mathsf{p}} \mathsf{S}_j$ to the plan, and order $(\mathsf{S}_i < \mathsf{S}_j)$.

3. If the choice is a link $\mathsf{S}_i \xrightarrow{\mathsf{p}} \mathsf{S}_j$ threatened by some consequence of a step S_t, then choose to either

   - (demote) add the ordering $(\mathsf{S}_t < \mathsf{S}_i)$ to the plan,
   - (promote) add the ordering $(\mathsf{S}_j < \mathsf{S}_t)$ to the plan,
   - (confront) choose some consequence κ of S_t that does *not* contain $\overline{\mathsf{p}}$ and adopt the trigger of $\mathsf{S}_{t\kappa}$ as a subgoal, or
   - (branch) restrict the context of S_t so it is incompatible with either the context of S_i or S_j.

3.4 Example

Recall that the example problem consists of

- An initial probability distribution over states, which is converted to an initial plan step (Figure 4).
- A goal expression {PR, PA, $\overline{\mathsf{ER}}$} (the widget should be processed and painted, and the plan should not have an error). The goal expression is converted to a final plan step (again Figure 4).
- A set of actions: paint, reject, ship, and inspect (Figures 1, 2, and 3.)
- A success threshold of $\tau = 0.8$.

The initial set of subgoals consists of the goal propositions: $\{\langle \mathsf{PR}, \mathsf{goal}\rangle, \langle \mathsf{PA}, \mathsf{goal}\rangle, \langle \overline{\mathsf{ER}}, \mathsf{goal}\rangle\}$.

Suppose the planner makes the following choices. First it supports $\langle \overline{\mathsf{ER}}, \mathsf{goal}\rangle$ with a link from consequence α of initial. Next it supports $\langle \mathsf{PA}, \mathsf{goal}\rangle$ by adding a step paint, linking its α consequence to the goal, thus adopting $\langle \overline{\mathsf{PR}}, \mathsf{paint}\rangle$ as a subgoal, since $\overline{\mathsf{PR}}$ is a trigger for the α consequence. This pair can in turn be supported by a link from the initial step. Now $\langle \mathsf{PA}, \mathsf{goal}\rangle$ has probability 0.95 of being true but the plan has success probability 0 because PR is false.

Next the planner supports $\langle \mathsf{PR}, \mathsf{goal}\rangle$ by adding a ship step, linking its α consequence to the goal, and as a result two new pairs, $\langle \overline{\mathsf{FL}}, \mathsf{ship}\rangle$ and $\langle \overline{\mathsf{PR}}, \mathsf{ship}\rangle$, become subgoals. Both of these pairs can be linked to the initial step's α consequence.

The resulting partial plan has three threats:

- The link $\mathsf{initial}_\alpha \xrightarrow{\overline{\mathsf{PR}}} \mathsf{paint}$ is threatened by ship, in particular its α and β consequences. (In other words, ship might make PR true, but paint requires it to be false.)
- The link $\mathsf{initial}_\alpha \xrightarrow{\overline{\mathsf{ER}}} \mathsf{goal}$ is threatened both by paint (consequence γ) and by ship(consequences β and γ).

The first threat can be resolved by *demoting* the ship step, ordering it after paint. The second two threats cannot be resolved by adding ordering constraints: no step can be ordered before the initial step or after the goal step, so these two threats must be confronted or branched. Suppose the planner decides to use confrontation. The planner chooses non-threatening consequences for each step: ship_α and paint_α, notes in the links that the threat was confronted ($\mathsf{initial}_\alpha \xrightarrow[\mathsf{ship}_\alpha, \mathsf{paint}_\alpha]{\overline{\mathsf{ER}}} \mathsf{goal}$),

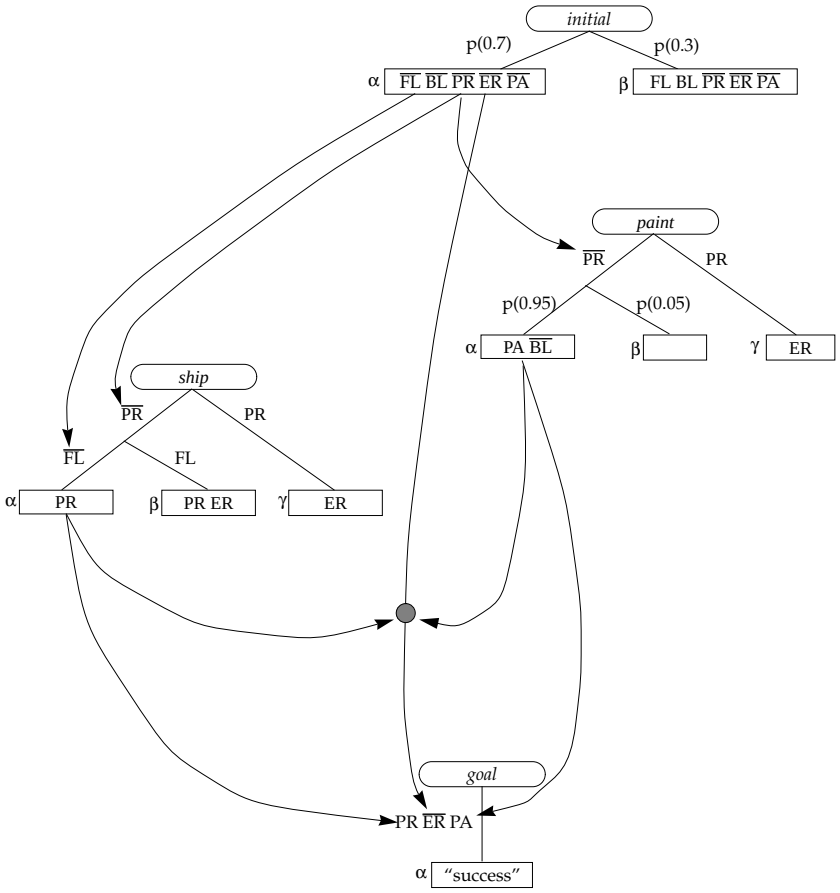

Figure 5: A plan with success probability 0.665

and adopts as subgoals the triggers of the non-threatening consequences: $\langle \overline{\mathsf{PR}}, \mathsf{ship} \rangle$, and $\langle \overline{\mathsf{PR}}, \mathsf{paint} \rangle$ (as it happens, both were subgoals already).

Figure 5 shows the resulting plan. Temporal ordering among the steps goes from top to bottom: $\mathsf{initial} < \mathsf{paint} < \mathsf{ship} < \mathsf{goal}$, links appear as arrows from consequence boxes to trigger propositions, and confronted threats appear as arrows from the non-threatening consequence (e.g. ship_α) to the threatened link ($\mathsf{initial}_\alpha \xrightarrow[\mathsf{ship}_\alpha, \mathsf{paint}_\alpha]{\overline{\mathsf{ER}}} \mathsf{goal}$).

This plan, which is the one that the non-contingent planner BURIDAN would produce, will work just in case the widget is initially not flawed and the paint step works, which translates into a success probability of $(0.7)(0.95) = 0.665$. The success probability can be increased somewhat by adding additional paint steps to raise the probability that PA will be true, but without introducing information-producing actions and contingent execution no planner can do better than 0.7.

At this point a reasonable refinement to the plan in Figure 5 would be to support the pair $\langle \mathsf{PR}, \mathsf{goal} \rangle$ by adding a reject step and linking it to the goal. The problem with this strategy is that it introduces a pair of irreconcilable threats: reject makes PR true, which threatens the link from initial to ship, and likewise ship makes PR true, threatening a link from initial to reject. Adding orderings can resolve only one of these two threats, and confronting the threat means that the planner will be trying to make reject's α consequence come true (so it will produce PR for the goal)

and simultaneously trying to make reject's γ consequence come true (so it won't produce PR before ship is executed). It cannot do both successfully, since any two consequences of a single action are mutually exclusive, so decreasing the probability of one threat increases the probability of the other. This plan appears in Figure 6, with threatened links appearing in grey. The solution is to execute one or the other (but not both) depending on the state of FL.

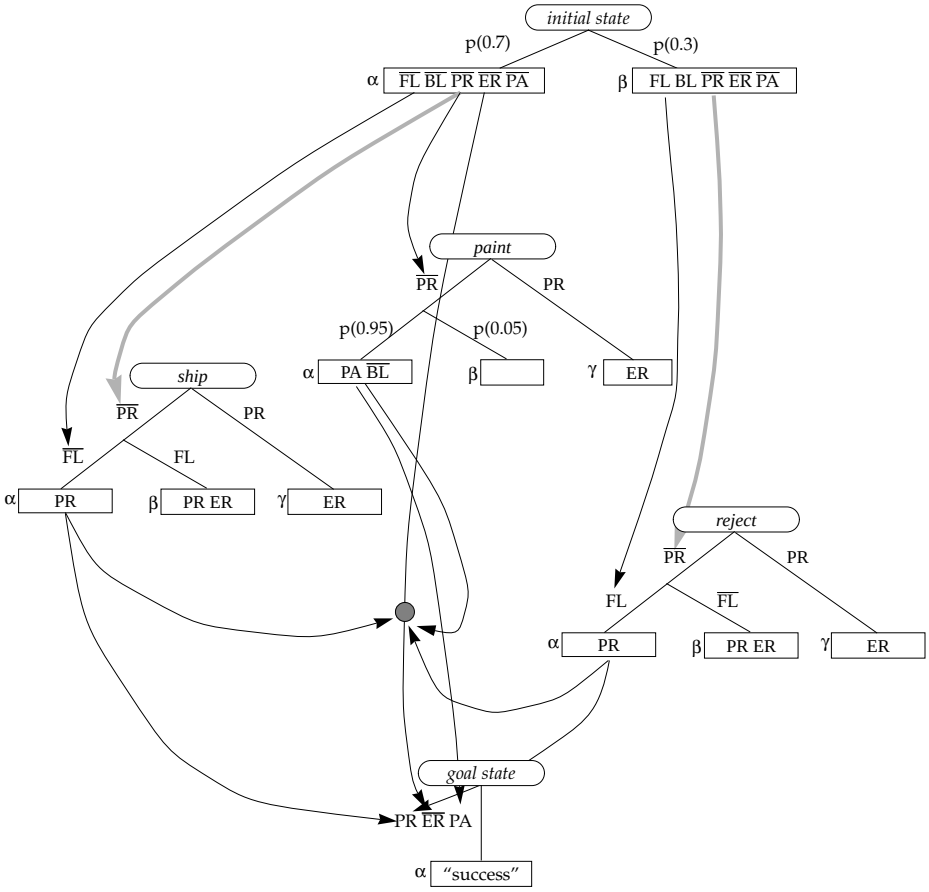

Figure 6: ship and reject threaten each other

Here the need for information gathering and conditionals becomes apparent: the planner needs to be able to insert steps that provide information about the state of FL and needs to be able to make the steps executed in the plan depend on that information. We next introduce a method for resolving threats based on making the context of the participating steps incompatible, i.e. by making context(S_i) $\wedge$ context(S_j) $\wedge$ context(S_t) unsatisfiable.

3.4.1 Threat resolution by branching

We call this new threat-resolution strategy "branching" because we will introduce contingency branches into the plan that prevent the threat from materializing.[5] Branches are a new element in

[5][Peot and Smith, 1992] refers to this approach as "conditioning." We adopt an alternative term because of a possible confusion with the use of the term in probabilistic reasoning, e.g. "conditioning on new evidence."

a plan's structure, analogous to causal links. A branch connects an information-producing step to a subsequent step, indicating the observation labels of the first that enable execution of the second. We will add two branches to our example plan: inspect=$\{$**ok**$\}\Rightarrow$ship and inspect=$\{$**bad**$\}\Rightarrow$reject. The first means that ship should be executed only if the execution of inspect generates an observation label of **ok**, the second means that reject should be executed only if the execution of inspect generates an observation label of **bad**. These restrictions are realized by adding the appropriate observation label to the steps' contexts.

The general procedure for resolving a threat S_t to a link $\mathsf{S}_i \xrightarrow{\mathsf{p}} \mathsf{S}_j$ by *branching* is as follows:

1. Choose to make context(S_t) incompatible with either context(S_i) or context(S_j). Let S_s be the step chosen.

2. Choose some information-producing step S_p that can be ordered both before S_s and before S_t. S_p must also be executable in some context compatible with both S_s and S_t, that is, context(S_p)$\wedge$context(S_s)$\wedge$context(S_t) must be satisfiable. S_p need not be in the plan already—it can be added to the plan in order to resolve the threat.

3. Choose a set $c = \{c_1, \ldots c_n\}$ and a disjoint set $c' = \{c'_1, \ldots c'_m\}$ of S_p's observation labels.

4. Add ordering constraints to the plan: ($\mathsf{S}_p < \mathsf{S}_s$), ($\mathsf{S}_p < \mathsf{S}_t$)

5. Add branches to the plan: S_p=$c\Rightarrow\mathsf{S}_t$ and S_p=$c'\Rightarrow\mathsf{S}_s$

6. Make step contexts more restrictive:

   - context(S_t) := context(S_t) $\wedge$ ($c_1 \vee c_2 \ldots \vee c_n$)
   - context(S_s) := context(S_s) $\wedge$ ($c'_1 \vee c'_2 \ldots \vee c'_m$)

7. Add to the plan's subgoals all pairs of the form $\langle \mathsf{p}, \mathsf{S}_p \rangle$, where p is any proposition in any of S_p's triggers.

3.5 Example

Return now to the example which progressed as far as Figure 6—the planner had added both ship and reject steps in order to raise PR's probability over 0.7. Doing so led to a threat (both ship and reject require PR to be false initially and make it true) that could not be resolved by ordering or by confrontation. This threat can be solved by branching, however. Suppose that the planner first notices that the reject step is threatening the link of the form $\mathsf{initial}_\alpha \xrightarrow{\overline{\mathsf{PR}}} \mathsf{ship}$.

1. It chooses to make the contexts of the link's consumer ship and the threatening step reject incompatible.

2. It chooses an information-producing action inspect and adds it to the plan.

3. It chooses subsets of inspect's observation labels: $c = \{$**bad**$\}$, $c' = \{$**ok**$\}$.

4. It adds ordering constraints to the plan, (inspect $<$ reject), (inspect $<$ ship), ensuring that the information-producing step will be executed before the actions whose contexts depend on its observation label.

5. It adds branches inspect={**ok**}⇒ship and inspect={**bad**}⇒reject to the plan.

6. It restricts the steps' contexts as follows:

   - context(reject) := $\mathsf{T} \wedge \{\mathbf{bad}\} = \mathbf{bad}$
   - context(ship) := $\mathsf{T} \wedge \{\mathbf{ok}\} = \mathbf{ok}$

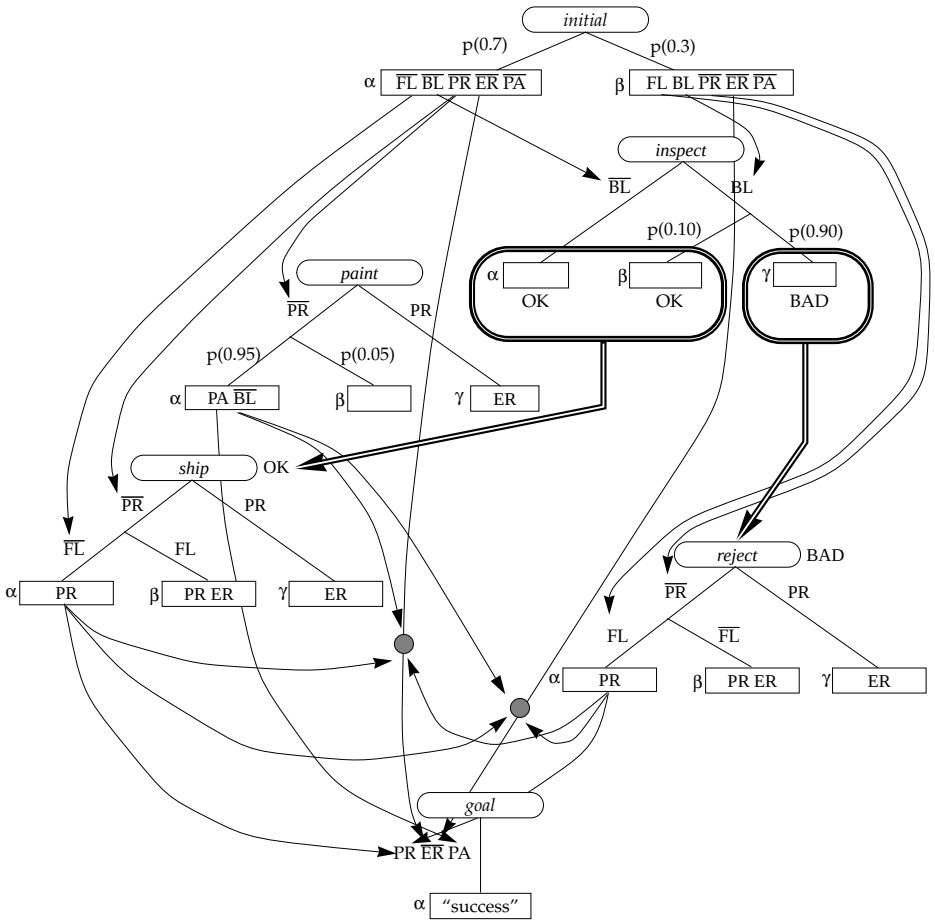

Figure 7: A successful plan

Now ship and reject are restricted to mutually exclusive execution contexts, and the plan is shown in Figure 7.

The resulting plan admits two execution sequences:

- inspect, then paint, then ship or reject,
- paint, then inspect, then ship or reject.

The first sequence has success probability .9215: it will fail only if the paint step fails or if the widget was blemished and the inspect step incorrectly reports **ok**. The second sequence has success probability 0.665, however: when paint is executed it makes BL false with probability .95, (and with

probability 0.05 it fails to make PA true, which means the plan will fail anyway). In any execution path in which paint succeeds, it will also make BL false, which means that inspect will certainly report **ok**, and the widget will be shipped regardless of whether or not it was initially flawed.

The assessment algorithm in Sections 3.3.1 and 2.6 will find and return the high-probability sequence, and the planner will terminate successfully. It is interesting to note, however, that the planner could further refine this plan, adding an ordering constraint so *only* the high-probability sequence is valid. How can the planner recognize and eliminate the bad sequence? Recall that all the trigger propositions of the inspect step are subgoals at this point—in particular, the planner can choose to support $\langle$BL, inspect$\rangle$ with a causal link from the initial step's β consequence. Then paint will threaten the link, since its α consequence makes BL false. One way this threat can be resolved is by ordering paint to come after the link's consumer, ordering it after inspect. The resulting plan has only one consistent ordering: inspect then paint then ship or reject, which has success probability 0.9215.

3.6 Contexts and plan structures

At this point we should mention a novel feature of our method of inducing conditional plan execution using restricted step contexts introduced by branching: the fact that our algorithm can generate plans whose executions "branch" then "rejoin." Suppose, for example, that we added an additional goal condition to the plan, that the planner notify a supervisor when it finished processing a widget. Assume that notify's single precondition is that the widget be processed (PR). C-BURIDAN can handle this easily, inserting the notify step along with a link from its single outcome to the goal step, then creating links from both ship and reject's α consequences to its trigger condition PR (see Figure 8). Note that since notify's execution context is T because it participates in no threats, so the execution sequence represented by this plan is inspect, paint, either ship or reject, notify. Other conditional planners, e.g. CNLP and Warplan-C [Warren, 1976] generate a new plan branch each time a conditional is inserted in the plan, and do not allow the branches to contain common steps.

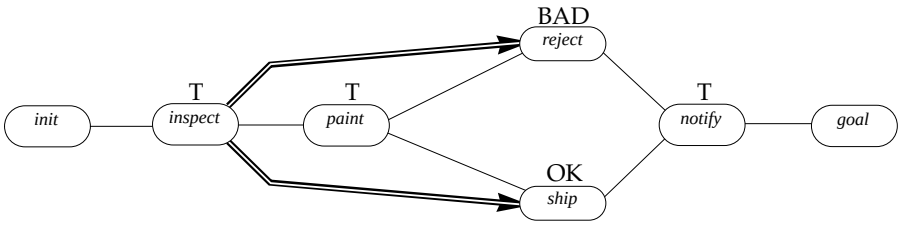

Figure 8: A plan with merged branches

4 Propagation of Step Contexts

The C-BURIDAN planning algorithm described in the previous section restricts a step's context only when doing so is necessary to resolve a threat. This policy produces correct plans, but admits inefficiency. Suppose, for example, that the ship action had an additional precondition HB (have box), and the planner had an additional operator get-box that had no preconditions and made HB true. At some point the planner would insert a get-box step in the plan and link it to ship, resulting in the situation pictured in Figure 9(a).

Notice that get-box's context will be unrestricted since it causes no threats nor is its link threatened by any other steps. Get-box will therefore be executed in all contexts, including those contexts in which reject will be executed instead of ship. We might want to restrict the context of this step so it is executed only when it is "useful," which in this case means in exactly those contexts in which ship is executed.

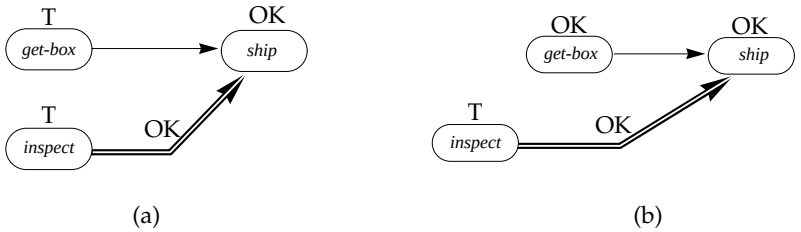

Figure 9: Restricting *have-box*'s context so that it is only executed when needed.

Since a step's reasons for being in a plan are recorded in the plan's link structure, we look there to determine when a step is useful. A step S is useful in a plan only when:

- At least one of the steps that produces propositions it consumes is executed, and
- At least one of the steps that consumes propositions it produces is executed.

If the first condition is not true, no previous step in the plan will produce the state that S requires to be effective, and since S can't do anything useful it might as well not be executed. If the second condition is not true then no subsequent step will be able to exploit whatever effects S was put in the plan to accomplish, therefore it might as well not be executed.
If we allow a step's context to be an arbitrary boolean expression (rather than a simple conjunction as assumed previously), we can constrain a step's context as follows:

- A step's context should at least be restricted to the disjunction of the contexts for all the steps that act as producers of causal links consumed by that step.
- A step's context should at least be restricted to the disjunction of the contexts for all the steps that act as consumers of causal links produced by that step.

The first constraint is enforced by a *forward propagation* of contexts through the plan whenever a new link or branch is added. The second constraint is enforced by a *backward propagation* sweep through the plan before it is returned as a solution. The second constraint allows us to restrict the context of get-box to be only **ok** (Figure 9(b)), since its only link in the plan is to ship, and ship's context is restricted to circumstances in which inspect reports **ok**.

4.1 Forward propagation

Forward propagation enforces the first constraint by computing a step's context as follows:

$$\begin{array}{lrcl} \text{let} & B & = & \text{the set of branches pointing to S,} \\ & C & = & \text{the set of consequences of S,} \\ & T(c) & = & \text{the set of trigger conditions for consequence } c, \end{array}$$

$$P(t) = \text{the set of steps producing links to trigger condition } t,$$

then,

$$context(\mathsf{S}) = (\bigwedge_{b \in B} branchlabel(b)) \wedge (\bigvee_{c \in C} \bigwedge_{t \in T(c)} \bigvee_{p \in P(t)} context(p))$$

where $branchlabel(b)$ is the set of observation labels associated with branch b.

The first clause represents the context information dictated by S's immediate branches (i.e. those assigned when it was involved in threat resolution by branching). The second clause formalizes the definition above: it is the disjunction of the contexts of all steps that produce propositions consumed by S's trigger propositions.

The context of a step S is recomputed whenever a new link or branch is made to it, and changes to its context are propagated recursively to the consumers of all links for which S acts as producer.

Notice that adding a new causal link to S can actually "widen" S's execution context (i.e. the step can be executable in more contexts as it consumes more links), and when a step's context widens it can threaten links that it did not threaten before. So whenever a step's context is changed, the algorithm must also check to see whether the step poses new threats to any links currently in the plan.

4.1.1 Empirical implications of context propagation

We noted initially that context propagation was not necessary to produce successful plans—foregoing the propagation phase means at worst that some steps might be executed unnecessarily in some contexts. We then introduced the propagation algorithm that is run every time a link or branch is added to a plan, one that requires traversing the plan's link structure and scanning it for new threats as well. The question therefore arises whether this additional computational step is worth the effort—perhaps the forward propagation should be performed only once, after a successful plan is built, to make sure its steps are executed only when necessary.

While it is an open empirical question as to whether frequent context propagation is a good idea, we should point out an additional benefit of context propagation: its potential to make the plan-generation process itself more efficient. Recall that propagation serves to restrict a step's context, and that a step can threaten a link only if its context is compatible with the link's context. Therefore restricting a step's context can serve to eliminate threats that would otherwise have to be resolved by the planner.

4.2 Reverse propagation

Forward propagation ensures that an action will only be executed when its causal links can in fact provide support for those consequences necessary for plan success. Reverse propagation, on the other hand, ensures that an action will only be executed when one of its consequences is actually needed—the context of a step S is conjoined with disjunction of the contexts of S's consumers.

There is an added restriction however: if S's context expression mentions an observation label **L**, then S must be ordered after the step that generates **L**. For example, in Figure 9, get-box must be ordered after inspect, since its new context is derived from that step. If the plan's ordering constraints do not permit this ordering, then all occurrences of the observation label **L** must be removed from S's context.

The algorithm for reverse propagation is therefore as follows:

1. Let $< \mathsf{S}_1, \ldots \mathsf{S}_n >$ be a sequence of steps representing a consistent ordering of the steps in plan $\mathcal{P}$

2. Loop for $\mathsf{S} = \mathsf{S}_n, \mathsf{S}_{n-1}, \ldots, \mathsf{S}_1$:

   (a) Let C be the set of steps that consume either links or branches from S.

   (b) $\text{context}(\mathsf{S}) := \text{context}(\mathsf{S}) \wedge \text{clean}(\bigvee_{c \in C} \text{context}(c))$

   (c) Order S after all the steps that generate observation labels that appear in context(S)

where clean(*expr*) removes from *expr* all the observation labels produced by steps that cannot be ordered before S.

4.2.1 Reverse propagation and completeness

One big difference between forward and reverse context propagation is that the latter actually adds ordering constraints to the plan. Forward propagation restricts a step's context, but does not add additional structure (links or orderings) to the plan.

Ordering constraints are added during plan refinement when links are created and when threats are resolved—both are necessary to preserve the probabilistic dependency between a link's producer and its consumer. Reverse context propagation, on the other hand, can order a step S_p after a sensing step S_s not because S_p's execution should directly depend on the consequent of S_s, but because S_p is a link producer for some third step S_c whose execution *does* depend on the result of S_s. Ordering S_p after S_s prevents the planner from later deciding that S_p should be ordered before, so after reverse propagation S_p could never produce a link consumed by S_s or any prior step even though semantically there is no reason why it could not do so. A planner that performs reverse propagation on every partial plan (i.e. after every plan refinement) is therefore incomplete.

Once again the question of whether and when to do reverse context propagation is an open empirical question—one can either perform reverse propagation on every partial plan and accept the incomplete algorithm as a concession to efficiency, or alternatively a planner could run the reverse propagation algorithm only once as a post-processing step, once a successful plan has already been found. Reverse propagation does not change the plan's probability of success, so it cannot make a successful plan unsuccessful.

5 Summary and Related work

C-BURIDAN is an implemented algorithm for plan generation that combines information-producing actions with a framework that allows the conditional execution of actions. Novel features of the representation include:

- The representation allows planning with actions whose effects and information content depend on both the prevailing world state and random chance.

- The action representation can encode a variety of senor models, including anything from asymmetrically noisy sensors to sensors that provide complete and flawless information about the world.

- The representation for information-producing actions is a natural extension of symbolic planning operators, yet admits a traditional Bayesian semantics for representing the information content of an action and for updating belief on the basis of observation.
- Causal and informational effects of an action can be combined arbitrarily—there is no distinction made between sensing and effecting actions.
- The framework allows reasoning about informational dependencies: the fact that an action can provide indirect information about the state of the system, and the fact that other actions can create or destroy dependencies among state variables.
- Our approach to restricting step contexts does not require tree-structured plans—plans can diverge initially in their execution contexts, then rejoin to execute steps that are needed in several contexts.
- While we have not proven that the C-BURIDAN algorithm is complete, it can solve problems that stumped BURIDAN and ones that CNLP could not solve.

5.1 Related work

Related work in conditional planning includes work in decision analysis as well as previous probabilistic planners and AI work on (deterministic) conditional planning.

5.1.1 Decision analysis

The concept of planning to gather information (and assessing the value of that information) is a common topic in the Decision Analysis literature, particularly the work on sequential decision making [Winkler, 1972]. Our approach uses the same Bayesian framework, but the emphasis is different in that we take our task to be one of *building* a good plan automatically from schematic action descriptions and an input problem. Work in decision analysis is generally more concerned with structuring the problem, eliciting domain and preference models from experts, and evaluating alternatives provided by experts rather than on algorithmic approaches to generating those alternatives.

Graphical structures like influence diagrams can be used to solve sequential decision-making problems, including those that allow information-gathering steps [Matheson, 1990]. We do not consider influence diagrams a solution to the *planning* problem in and of themselves, however—they do not address the problem of generating plans from action schemas and problem descriptions.

5.2 Probabilistic planning

Our work extends the BURIDAN planner [Kushmerick *et al.*, To appear]: we added to BURIDAN the idea of information-producing steps and step contexts, as well as the threat-resolution technique of branching (which is due to CNLP, see below), and the algorithms for context propagation. See [Kushmerick *et al.*, To appear] for a discussion of related probabilistic planning algorithms.

5.3 Conditional planning

Our approach to contingent planning borrows much from the CNLP algorithm of [Peot and Smith, 1992]. In particular we adapted their method of threat resolution by conditioning: that information-producing actions generate execution contexts, and that one way to resolve a threat is to force the threatening step and the threatened link to appear in mutually exclusive contexts.

CNLP and C-BURIDAN use different underlying representations. CNLP adopts a standard STRIPS (logical) semantics, whereas our C-BURIDAN n manipulates a probabilistic representation. Our inspect action would look like this in CNLP:

```
Name: Observe-widget
Pre:  Unk(blemished)
      (not (processed))
+a1:  (blemished)
+a2:  (not (blemished))
```

(It seems as though the first precondition, that the state of `blemished` be unknown, is not necessary—it indicates that it will never be *useful* to insert a step into a plan if (1) its only effect is to provide information about a proposition and (2) the state of that proposition has already been determined previously in the plan.)

Observation labels in C-BURIDAN play the same role as in CNLP—the annotations `+a1` and `+a2` above represent reports the sensor might produce at execution time, which are used in building execution contexts for subsequent steps.

CNLP's action model cannot represent a situation in which an operator performs differently depending on the prevailing world state or on unmodeled (chance) factors. CNLP therefore cannot model sensing operators like our inspect operator, whose behavior is state dependent and noisy. The fact that actions have multiple possible consequences complicates the planning process somewhat: since a link to a proposition does not guarantee the proposition's truth, we have to admit the possibility of multiple (disjunctive) causal support for propositions, and a step's context can contain disjunctions as well (whereas in CNLP a step's context is a conjunction of observation labels).

Our algorithm for introducing contingencies differs from CNLP as well: CNLP's approach is based on the idea that every time a new execution context is introduced into the plan (by conditioning or branching) a new instance of the goal step is also added with that context. The planner thus tries to build a plan that satisfies *all* instances of the goal step, i.e. a plan that will work under all circumstances. The resulting plans are tree-structured: once two branches in the plan are created to condition on possible sensor reports, subsequent steps in separate branches cannot share steps. We noted in Section 3.6 that our algorithm can represent plans in which certain steps (like ship and reject) should be executed in restricted execution contexts, but that subsequent steps (like notify) should be executed unconditionally.

5.4 Future work

Our preliminary work on C-BURIDAN has produced a good framework for posing the problem of conditional and informational planning, but the system is limited both by its representation and by its computational efficiency. Future work in the following areas is therefore indicated:

Computational properties Our initial experiments with BURIDAN showed us that the computational complexity of building probabilistic plans is significantly worse than building plans using a logical representation language. And the problem becomes worse still when the planner can insert information-gathering actions and conditionals. To put it bluntly, while C-BURIDAN is fully implemented, significant search control knowledge is necessary to enable solution of even simple examples like the one presented in this paper.

Consider the question of when it might be advantageous to insert an information-producing action into a plan. It can be difficult to ascertain exactly what propositions a particular sensing action provides information about. Recall from the example above that the inspect operator provides direct information only about whether the widget is *blemished*, but the planner really cares only about whether the widget is *flawed*. Information about BL turns out to convey information about FL *in this particular case*, but the relationship depends on the particular planning problem and indeed even the particular plan being constructed—the information content of an sensing operator cannot be determined by examining the operator alone. The basic computational problem is that *any* information-producing action can potentially provide information about *any* proposition. A practical planner will need fast heuristic methods for deciding what information it needs to gather in order to build a successful plan, and what information-producing operators are likely to provide it with that information.

More expressive languages C-BURIDAN's utility is limited by its propositional representation language. Related work is concerned with building and reasoning about plans using more expressive representation languages: [Hanks, 1993] explores the problem of assessing a plan's quality, but using a probabilistic framework that allows reasoning about sets and quantities. [Golden *et al.*, 1994] is an effort to incorporate a richer sensing model, based on UWL [Etzioni *et al.*, 1992], into the UCPOP partial-order planner. Its inference rules allow effective reasoning about locally complete information, enabling the planner to satisfy universally quantified goals under incomplete information, and to eliminate redundant sensing operations.

Value of information C-BURIDAN generates a plan with a particular probability of success. It is more common to analyze both plans and information-producing actions within plans in terms of their utilities or values: a value or utility model associates a value with a plan, then an information-producing action can be evaluated in terms of the value it contributes to a plan. [Matheson, 1990] demonstrates how to evaluate plans in this way, but does not provide an algorithm for generating plans. [Haddawy and Hanks, 1993] point out that planning to maximize the probability of goal success corresponds to planning to maximize expected value only for a particular extremely restricted class of utility models. C-BURIDAN must therefore be extended to generate plans according to a criterion of expected-utility maximization, at which point information-gathering actions can be evaluated in terms of the utility they add to the plan. We intend to apply the framework for utility models developed in [Haddawy and Hanks, 1993] to this planner, but doing so first requires extensions to the representation language as noted above.

Formal properties We need to complete proofs that the (nondeterministic) C-BURIDAN algorithm is sound and complete. We can use the formal framework developed to prove these properties for non-contingent BURIDAN, but need to extend the plan semantics to account for information-

gathering actions and branches. Previous work [Etzioni *et al.*, 1992] contains a compatible semantics for contingent plans, but uses a non-probabilistic language.

We also need to explore the relationship between our algorithm and algorithms from the decision sciences like value iteration [Howard, 1960] and other dynamic programming approaches [Raiffa, 1968].

References

[Allen *et al.*, 1990] J. Allen, J. Hendler, and A. Tate, editors. *Readings in Planning.* Morgan Kaufmann, San Mateo, CA, August 1990.

[Etzioni *et al.*, 1992] O. Etzioni, S. Hanks, D. Weld, D. Draper, N. Lesh, and M. Williamson. An Approach to Planning with Incomplete Information. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, October 1992.

[Golden *et al.*, 1994] K. Golden, O. Etzioni, and D. Weld. XII: Planning for Universal Quantification and Incomplete Information. Technical report, University of Washington, Department of Computer Science and Engineering, January 1994.

[Haddawy and Hanks, 1993] Peter Haddawy and Steve Hanks. Utility Models for Goal-Directed Decision-Theoretic Planners. Technical Report 93–06–04, Univ. of Washington, Dept. of Computer Science and Engineering, September 1993.

[Hanks, 1993] Steve Hanks. Modeling a Dynamic and Uncertain World II: Action Representation and Plan Evaluation. Technical report, Univ. of Washington, Dept. of Computer Science and Engineering, September 1993.

[Howard, 1960] Ronald A. Howard. *Dynamic Programming and Markov Processes.* MIT Press, 1960.

[Kushmerick *et al.*, To appear] N. Kushmerick, S. Hanks, and D. Weld. An Algorithm for Probabilistic Planning. *Artificial Intelligence*, To appear.

[Matheson, 1990] James E. Matheson. Using Influence Diagrams to Value Information and Control. In R. M. Oliver and J. Q. Smith, editors, *Influence Diagrams, Belief Nets and Decision Analysis*, pages 25–48. John Wiley and Sons, New York, 1990.

[Pearl, 1988] J. Pearl. *Probablistic Reasoning in Intelligent Systems.* Morgan Kaufmann, San Mateo, CA, 1988.

[Peot and Smith, 1992] M. Peot and D. Smith. Conditional Nonlinear Planning. In *Proc. 1st Int. Conf. on A.I. Planning Systems*, pages 189–197, June 1992.

[Raiffa, 1968] Howard Raiffa. *Decision Analysis: Introductory Lectures on Choices Under Uncertainty.* Addison-Wesley, 1968.

[Warren, 1976] D. Warren. Generating Conditional Plans and Programs. In *Proceedings of AISB Summer Conference*, pages 344–354, University of Edinburgh, 1976.

[Winkler, 1972] Robert L. Winkler. *Introduction to Bayesian Inference and Decision.* Holt, Rinehart, and Winston, 1972.