Issues in Using Scenarios in Deficiency-Driven Design

CIS-TR-92-19

Brian Durney
October 1, 1992

Department of Computer and Information Science
University of Oregon

Issues in Using Scenarios in Deficiency-Driven Design

Brian Durney

Department of Computer and
Information Science
University of Oregon
Eugene, Oregon 97403

durney@cs.uoregon.edu

ABSTRACT

In this paper I propose research in using scenarios to evaluate design changes in deficiency-driven design. I describe a method of deficiency-driven design and show how scenarios can be used in that design method. I state a question of evaluation in terms of scenarios and examine some planning techniques that are useful in addressing that question.

Relevant planning techniques include analysis of event dependencies, abstraction, localization and decision-theoretic planning. I describe work by several researchers in these areas and discuss issues that arise in applying that work to design evaluation. In particular I look at the problem of evaluating changes in terms of their generality and interactions with previous changes.

1. INTRODUCTION

To see what scenarios are and how we would use them in design evaluation, let's suppose that we are designing the security facilities for a computer system. We can describe some of our design goals in terms of sequences of events that might or might not happen. I will refer to such sequences of events as "scenarios".

For instance, in one scenario a user named Abigail is added to the list of users, logs in, creates a file and changes it. We want users to be able to access their own files in our artifact, so we will call this a "good scenario." We hope that our design allows all good scenarios. On the other hand, another scenario shows how a user might guess someone else's password. We don't want users to be able to do this so we will call this a "bad scenario". We hope that our design prohibits all bad scenarios.

By describing designs in terms of whether or not they allow good and bad scenarios we have begun to use scenarios in design evaluation. I propose research that would extend this intuitive beginning and address open issues in the use of scenarios in design evaluation.

In this paper I focus on how results from planning research can be applied to design evaluation. Thus I do not compare different methods of design but instead look at how one particular design method can incorporate different planning techniques. Furthermore I limit my discussion of planning techniques to those that most readily apply to design evaluation.

I begin by defining scenarios and design evaluation. Next I look at planning techniques that relate to design evaluation, and finally I discuss issues that arise in applying those techniques to design evaluation.

In Section 2 I describe a representation for designs and design constraints. I define scenarios and briefly compare my proposed research to other work with scenarios. Next I describe a process of deficiency-driven design which uses scenarios as deficiencies. Finally I state the problem of design evaluation in terms of consequences of design changes: enabling and disabling scenarios and nonfunctional changes.

In Section 3 I relate the consequences of design changes to planning problems and describe some results of planning research that can be applied to design evaluation. These results are analysis of event dependencies, event abstraction hierarchies and expected utility of plans.

I identify issues in applying planning results to design evaluation in Section 4. The major issues are agents in the environment, adaptability, regression, likelihood and utility values, multiple perspectives and artifact costs vs. design costs.

In Section 5 I propose research on the question of evaluation posed at the end of Section 2. In particular I will investigate the use of abstraction and dependency analysis to address the issues of adaptability and regression in design evaluation. As part of this research I will create a program to evaluate design changes in the domain of computer security. For that reason, examples in this paper will be in the domain of computer security but they are meant to illustrate planning and design concepts rather than computer security concepts.

2. SCENARIOS IN DESIGN EVALUATION

To understand how scenarios can be used in design evaluation we must first define scenarios and see how they relate to design. We must also define the design process so that we can understand the role that evaluation plays in that process.

In the following subsections I'll give definitions and examples of designs, design constraints and scenarios, and then will describe the process of deficiency-driven design. I'll conclude this section by stating the question of evaluation in terms of enabling and disabling scenarios and in terms of relative evaluation of scenarios.

2.1 Designs

For purposes of this paper, the product of the design process is a description of the functionality of the artifact. I follow Anderson & Fickas [1989] in representing that functionality as a set of planning operators.

Operator Set

Add User (*user*)

Create (*user, file*)

Login (*user*)

Read (*user, file*)

Read PF(*user*)

Write (*user, file*)

Guess Password (*user1, user2*)

Figure 2.1: A design.

Figure 2.1 shows an example of a design. The design represents an artifact that allows agents to add user entries to password files, log in to the system, use programs to read and write files and so on.

In addition to describing the functionality of the artifact, the design in Figure 2.1 also describes functionality available in the artifact's environment. Environmental operators such as "guess password" are important because they allow the computer to reason about what agents in the environment can do and how they will interact with the artifact.

When the designer changes the design of the artifact by adding or deleting operators she changes the scenarios that occur with that design. To evaluate those changes she needs some statement of what scenarios are good or bad. The next two sections address this issue.

2.2 Constraints

Where a design (or artifact description) is the output of the design process, constraints are the input. The client who requests a design can tell the designer what he or she wants by stating constraints on the artifact. In deficiency-driven design with scenarios the client can state constraints in terms of desired or prohibited states [Anderson and Fickas 1989].

For example, in describing a computer system the client might prohibit states where one user has access to another user's files. The client might also want users to access their own files, so states where users can get to their own files are desired states. The client simply states these as constraints; the designer must determine how the design will satisfy the constraints. In deciding how best to satisfy the various constraints on the design, the designer can use scenarios to see how the current design relates to constraints.

2.3 Scenarios

I define a scenario as a sequence of events. Figure 2.2 shows an example of a scenario, where each box represents an event. An event is an action of some agent. Thus in figure 2.2, the first event is the action of adding Abigail's entry to the password file.

We can think of scenarios that show achievement of prohibited states, one type of constraint violation, as bad scenarios. Likewise, scenarios that show achievement of desired states are good scenarios. The designer attempts to satisfy as many constraints as possible by disabling bad scenarios and enabling good ones.

Add User (Abigail)
Login (Abigail)
Create (Abigail, File1)
Write (Abigail, File1)

Figure 2.2: A "good" scenario.

In my research, scenarios are stated formally; other research deals with informal scenarios, where an event is a natural-language description of some action or state change. For instance, Wexelblat [1987] describes scenarios as "a flexible, informal medium for carrying on a high-level conversation between groups of designers and users."

Although informal scenarios are useful in human communication when, as Wexelblat says, "customers and designers do not speak the same language," they are not useful in human-computer communication because most computer programs have no way of reasoning about natural-language descriptions of events.

Since computers can't reason about informal scenarios, some researchers use formal scenarios so that the computer can assist a human designer or user in analyzing a design or specification. Benner and Johnson [1989] describe how formal scenarios can be used in specification and Kaufman [et al, 1989] describes a rule-based system that helps a user analyze a specification in terms of scenarios that could happen given some initial state. SXL [Lee and Sluizer 1991] is another system that helps a user find out what scenarios are possible in some design.

To date, most work in scenarios has focussed on using scenarios to help a human designer understand a design. Anderson and Fickas [1989] and Fickas Helm [1991], however, have shown how a computer program can use scenarios to help construct a design. The research I propose builds on their work and in particular deals with some issues in scenario evaluation and design evaluation which they don't address.

Now that I've defined constraints, designs and scenarios, I'll discuss the design process and design evaluation.

2.4 The Design Process: Deficiency-Driven Design

Suppose that a design allows a bad scenario. That scenario is a *deficiency* in the design because it shows a violation of a constraint. If the designer discovers such a deficiency she will change the design in hopes of eliminating the deficiency.

The designer changes the design by adding or deleting operators from the operator set that represents the functionality of the design. Such changes will have effects on the design that

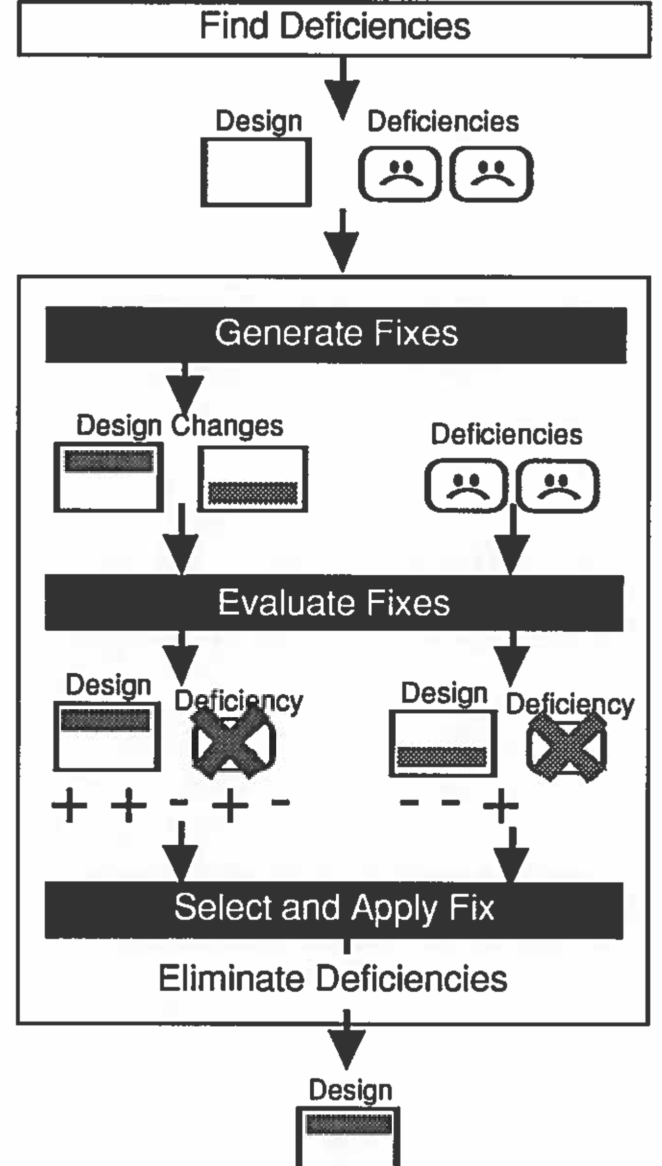

Figure 2.3: Deficiency-driven design using scenarios.

can be stated in terms of scenarios. For example, if the designer deletes the log in operator from the design of a computer system then all scenarios where users log in to the system will be impossible.

Deficiency-driven design is a design method wherein a designer executes (or simulates) behavior of an artifact and thereby finds problems (deficiencies) in the design. Kant [1985] describes the basic principles of the method and Kant and Steier [1985] describe its use in algorithm design.

Deficiencies drive the design because they not only show a problem that needs fixing but often suggest a solution. In that sense, deficiencies provide constructive criticism. As a deficiency, a scenario is constructive criticism because it indicates possible changes to the design. Given a bad scenario, the designer can delete any operator that the scenario requires and thus disable the scenario and eliminate the deficiency. Evaluation of the

various changes is still necessary but the scenario helps reduce the number of changes to consider.

Figure 2.3 shows the basic steps in deficiency-driven design. They are to find deficiencies and eliminate them.

The designer first finds a set of deficiencies. The deficiencies will be scenarios— either bad scenarios to disable, or good scenarios to enable. The design might have many known deficiencies in which case the designer simply selects a set for the current focus, or the design might have no known deficiencies so that finding a deficiency entails analysis to discover deficiencies.

The designer next eliminates one or more deficiencies. This step can be further broken down into three steps—generating fixes, evaluating fixes and selecting a fix.

First, the designer generates fixes— design changes intended to eliminate one or more deficiencies. Since the generator could produce numerous fixes, including many superfluous ones, the generator must be constrained.

Such constraints could be in terms of evaluation techniques applied in the generator. However, the evaluation issues I discuss in this paper arise regardless of whether they are applied in the generator or in a separate evaluator.

Second, the designer evaluates each alternative in terms of which scenarios it enables or disables and in terms of the significance of those scenarios. This paper discusses issues that arise in this step of the design process. In the next section I will look at this step in detail.

Third, the designer selects a fix based on the evaluation and applies it. In most design problems making the decision requires subjective judgment and so requires input from a human user.

The designer has now produced a new design and can repeat the process to further improve the design.

Each step of this design process requires information and techniques. I plan to focus on the evaluation step so I will now examine it in more detail.

2.5 The Question of Evaluation

Suppose a design has the deficiencies shown in Figure 2.4. The first deficiency is a scenario where one user guesses another user's password. In the second deficiency a user writes down his password and loses it. Another user finds the password. In the third deficiency a user cannot access her files.

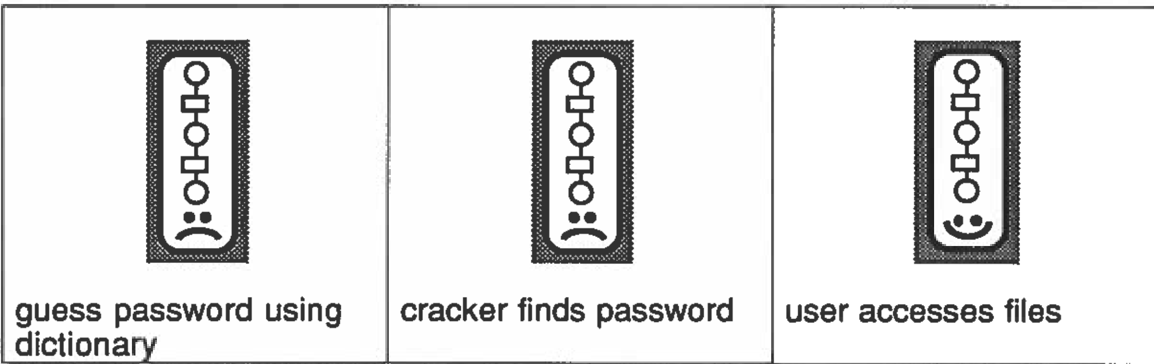

Figure 2.4: Three deficiencies. The first and second scenarios are deficiencies if enabled, the third scenario is a deficiency if disabled.

Suppose also that the designer (either human or automated) proposes three changes to the design as shown in Figure 2.5. The first alternative is to assign random passwords to the system users; the second is to let users choose their own but have the system check them, and the third is to require a dynamic signature check as part of the log-in procedure.

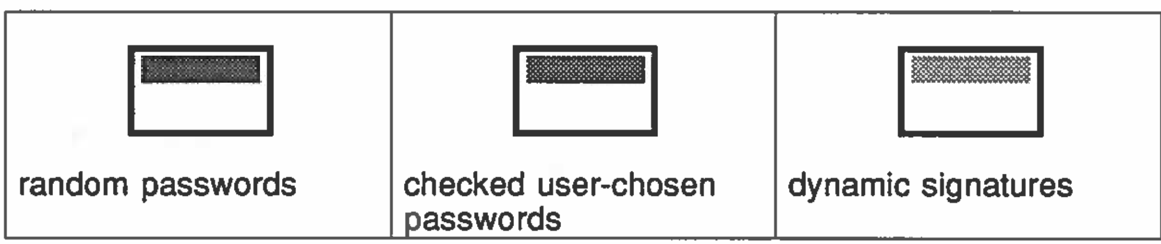

Figure 2.5: Three ways to change the design.

The designer can evaluate design changes by determining whether or not they eliminate given deficiencies. In this example, the designer can determine whether each of the changes in Figure 2.5 enables or disables the scenarios shown in Figure 2.4.

The first alternative, assigning random passwords, disables the guess password scenario because it is difficult to guess random passwords. Assigning random passwords enables the lost password scenario because users are more likely to have trouble remembering random strings of characters and are more likely to write them down. For similar reasons, the first alternative can prevent access to files because a user is likely to forget a random password and not be able to log in.

The second alternative, checking user-chosen passwords, also disables the guess password scenario, but does not enable the lost password or the forgotten password scenarios.

Finally, the third alternative, the dynamic signature check, disables the guess password scenarios but does not enable the second and third deficiencies. In fact, users are more likely to forget a user-chosen password than they are to "forget" their signature. However, this alternative requires expensive hardware that the other two choices do not.

Scenario / Design Change	guess password using dictionary	cracker finds lost password	user accesses files	$$$$
random passwords	DISABLES	makes more likely	makes less likely	NO CHANGE
checked user-chosen passwords	DISABLES	NO CHANGE	NO CHANGE	NO CHANGE
dynamic signatures	DISABLES	DISABLES	NO CHANGE	↑

Figure 2.6: Evaluation summary.

Figure 2.6 summarizes the evaluation example outlined in the preceding paragraphs. In this example, the random password change causes deficiencies that the other two alternatives do not, and the dynamic signature equipment is much more expensive. While a human designer would rely on additional information, this evaluation distinguishes between the choices and provides useful evaluation information by relating design changes to scenarios and thus to constraints.

Generalizing this example, we see two questions for evaluation using scenarios: Which scenarios are enabled or disabled by a design change? Which scenarios are more expensive or more likely? The second question arises because the designer might not be able to eliminate all deficiencies and so will want to leave enabled the scenarios which do the least harm or are the least likely, and will want to leave disabled the scenarios which do least benefit or are least likely.

I propose research that addresses the following question:

What information and methods are necessary for a program to evaluate design changes in terms of what scenarios they enable or disable, and in terms of the cost, benefit and likelihood of scenarios?

As a first step to answering this question I've looked at the literature of related fields. Since deficiencies are scenarios or sequences of events I've studied the planning literature and have found useful results. I'll now describe why the planning literature is relevant and what it has to offer in answering my design evaluation question.

3. USEFUL RESULTS FROM PLANNING

I concluded the previous section by stating the question of evaluation in terms of enabling and disabling scenarios and in terms of nonfunctional changes. In this section I'll look at what planning research offers to address that question.

A typical problem in planning is to find a plan that achieves a specified goal. An initial state and a set of actions or operators are also given. Operators are defined in terms of arguments, preconditions and effects. The complete plan is a sequence of operators where every operator has all of its preconditions satisfied and the final state meets the requirements stated in the goal.

Since a scenario is a sequence of events, it can be considered a plan. The designer can represent the scenario formally as a sequence of operators and can reason about whether various changes to the plan will disable a complete plan or enable an incomplete one.

In the following sections I will state the questions of enabling and disabling scenarios as problems in adaptive planning and counterplanning and the problem of nonfunctional changes as a problem in plan evaluation and see how planners deal with these problems. I'll describe three important concepts that planners use: event dependencies, abstraction and expected utility of plans.

Figure 3.1: Breaking a plan.

3.1 Fixing and Breaking Plans

A design change eliminates a deficiency if it enables a good scenario or disables a bad one. Plan repair in planning research deals with enabling plans, and counterplanning deals with disabling plans, so we can look for useful techniques for enabling and disabling scenarios in those areas of planning research.

As an example of counterplanning consider the plan shown in Figure 3.1. One way to make this plan fail is by realizing that the cracker's guess password action requires an easy-to-guess password. By changing the add user operator so that it adds random passwords, the planner can prevent the guess password operator from succeeding and can thereby break the plan. A designer can use similar techniques to disable a scenario.

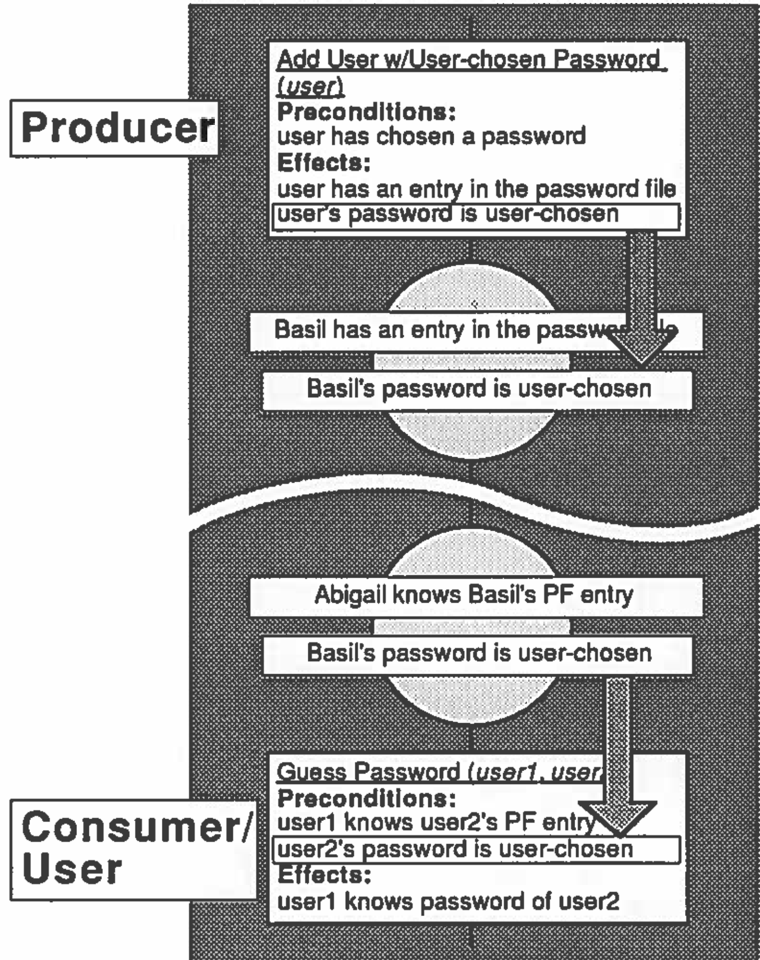

Figure 3.2: Event dependencies.

As an example of plan repair, consider how the cracker might repair this plan. Rather than using a dictionary to guess the password, the cracker might find a way of guessing the password that works for random passwords. For example, the cracker might encrypt all strings less than eight characters long and compare them to the entries in the password file. The cracker is repairing the plan or adapting it to new circumstances. A designer can use similar techniques to enable a scenario.

In this section I'll look at some planners that use counterplanning and plan repair and see how they use event dependencies and abstraction hierarchies.

Analysis of Event Dependencies

An event depends on some condition if the condition is one of its preconditions. I will refer to it as a *requirer* of that condition (or a *consumer* if it makes the required condition no longer true). The event that makes a condition true is a *producer* of that condition. Figure 3.2 illustrates a producer and a consumer.

A planner can enable a plan by adding a producer that makes some necessary condition true, or it can disable a plan by adding a consumer (i.e., a deleter). Similarly, it can disable

a plan by deleting a producer or by adding a consumer of some necessary condition (see Figure 3.3).

Whether or not a new event fixes or breaks a plan depends on whether it produces or consumes necessary conditions in the plan. Therefore if a planner can effectively reason about producer/consumer relationships and event dependencies it can evaluate a plan change as a functional fix. Use of event abstraction hierarchies helps planners reason about event dependencies and thus helps them evaluate plan changes.

Enable a Plan	Disable a Plan
Add a producer.	Add a consumer.
Delete a consumer.	Delete a producer.
Enable the precondition of a producer.	Enable the precondition of a consumer.
Disable the precondition of a consumer.	Disable the precondition of a producer.

Figure 3.3: Enabling and disabling plans.

Event Dependencies in PLEXUS

PLEXUS [Alterman 1988] is a planner that adapts plans to new situations. It represents some dependencies as *reason links* and uses them to repair plans. Reason links enable PLEXUS to determine what steps of the plan will be affected by failing steps and thereby to determine what changes to the plan will enable it.

Figure 3.4 shows some reason links in a plan that shows Abigail gaining access to Basil's secret information. Suppose that the log in step fails, so that Abigail is unable to log in. Since a reason link connects the Log In step to the Get PF Entry step, that step will also fail.

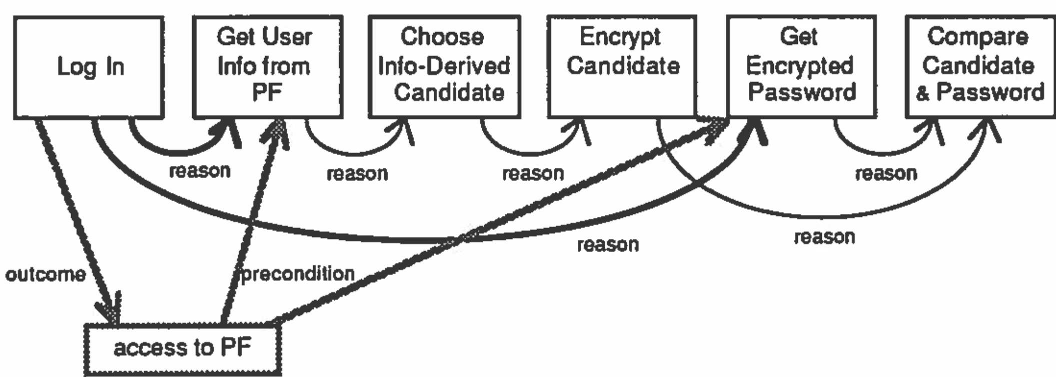

Figure 3.4: Some reason links in a PLEXUS-style plan.

Given a failing step such as this one, PLEXUS has two alternatives: find an alternative plan to establish the failing step's outcome, or find an alternative to the step that relies on the

failing step's outcome. In other words, it can look for alternatives to steps on either end of the reason link, as shown in Figure 3.5.

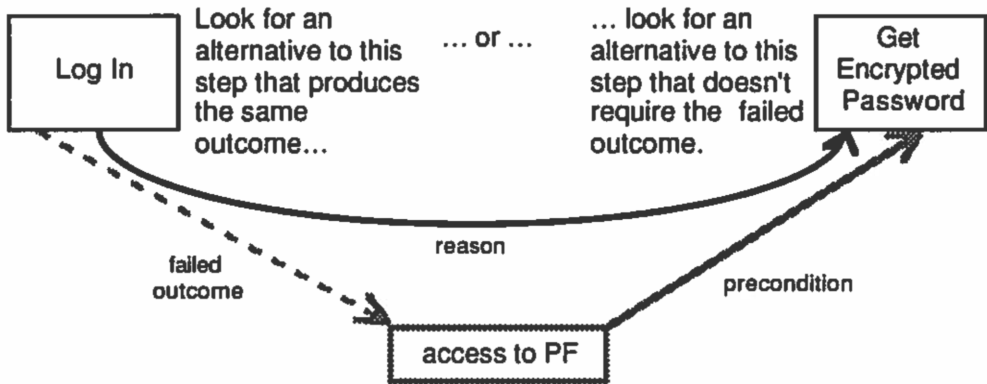

Figure 3.5: How reason links guide plan repair.

In this example, PLEXUS can find an alternative to Log In or an alternative to Get Encrypted Password. As an alternative to logging in and reading the password file, Abigail could try to get access to the file by anonymous FTP. Instead of using password file information Abigail could use other sources of information to generate and test passwords. In the section on abstraction I will describe how PLEXUS uses abstraction to find these alternatives.

Event Dependencies in POLITICS

POLITICS [Carbonell 1981], like PLEXUS, addresses issues in fixing plans. Where PLEXUS adapts plans to a new situation, POLITICS reasons about how to overcome active obstruction by another agent. Furthermore, POLITICS also takes the role of the obstructor and tries to break another agent's plans. Obstructing and overcoming obstructions to plans is what Carbonell calls *counterplanning*.

POLITICS must use event dependencies in its counterplanning strategies. For example, suppose that POLITICS (or a similar hypothetical program) were given the problem of guessing passwords as stated earlier. Using counterplanning techniques, POLITICS would find ways for Basil to obstruct Abigail's plans to access Basil's secret information. One way of doing so is to block some necessary precondition of Abigail's actions. Preconditions determine an action's dependencies, so POLITICS is thus reasoning about event dependencies.

Abstraction in Planning

Event dependency analysis is one technique that planners use to fix and break plans. Another technique is abstraction. In this section I describe four programs that reason about abstraction. Since they use abstraction they all have some way of ignoring details but their purpose in doing so, the information that they abstract away and their method of abstraction differ from one program to the next.

Abstraction in PLEXUS

PLEXUS [Alterman 1987] uses event abstraction hierarchies to adapt plans to situation differences. As described earlier, it uses event dependencies to determine what parts of a plan must change and then uses abstraction to find alternatives to those parts of the plan.

To continue the earlier PLEXUS example, once the planner realizes that it must change either the Log In step or the Get Encrypted Password step, it uses abstraction to find alternatives to those steps. Figure 3.6 shows part of an abstraction hierarchy similar to the ones that PLEXUS uses.

If PLEXUS decides to adapt the Log In step uses abstraction to find another step that produces the same desired outcome, i.e., access to PF (i.e., access to the password file). There are many possible abstractions for each plan step, so PLEXUS relies on the purpose link to help it select the right abstraction. In this case the purpose is to get access to the password file, so PLEXUS abstracts to Get Access to Password File and can then specialize to Anonymous FTP of Password File. It must then check to see if the preconditions of this step hold. If they do not, PLEXUS must search for additional alternatives.

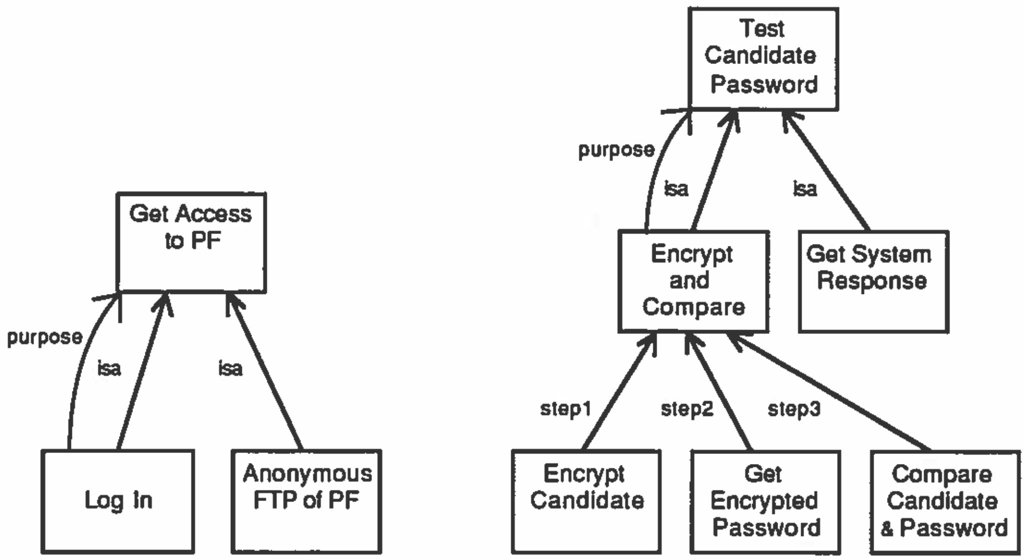

Figure 3.6: Parts of a PLEXUS-style abstraction hierarchy.

PLEXUS could also choose to modify later steps in the plan so that the failed step is unnecessary. It uses the reason links to determine what plan steps those are and then it abstracts on those steps.

Since the Log In step failed in this example, the access to PF precondition of Get Encrypted Password doesn't hold. Therefore, PLEXUS abstracts along the purpose link until it reaches a point where that precondition is abstracted away. It then tries to find a specialization where the preconditions do hold in the new situation. In this case, it abstracts to Test Candidate Password where PF access is not necessary and specializes to Get System Response which does not require access to the password file.

Abstraction in OPIE

OPIE [Anderson and Farley 1988, 1990] is another planner that uses event abstraction hierarchies. Given a set of operators, OPIE builds an object hierarchy and an event hierarchy which it uses during planning. To build the operator hierarchy, OPIE compares the precondition, add and delete lists of operators. Every pair of operators that share a common literal will have a common ancestor in the operator hierarchy.

During planning OPIE uses two basic constraints: there must be no unproduced conditions and there must be no abstract operators in the finished plan. In order to avoid an unproduced condition, OPIE adds an operator to the plan that will produce the condition. It always adds the most abstract operator possible. By selecting events in this way, OPIE avoids making unnecessary commitments that might have to be retracted later. In order to avoid violating its other constraint— no abstract operators in the finished plan— OPIE specializes operators as it acquires information that constrains the specialization.

The abstraction hierarchy determines what specializations are possible for a given abstract operator. In a sense, the abstract event stands in for all possible alternatives that enable the current plan. Thus abstraction allows the planner to analyze the number and kind of alternatives available for a given plan slot.

In many ways OPIE's use of abstraction is similar to that of PLEXUS. Both reason about abstract operators, producer/consumer links (called reason links in PLEXUS) and decomposing plans into steps.

There are also important differences between the two. Unlike PLEXUS, OPIE generates abstraction hierarchies. As an adaptive planner, PLEXUS reasons about what changes are necessary and uses an existing hierarchy but has no technique for automatically constructing the abstraction hierarchy.

Another difference is that OPIE has no previous concrete plan to reuse, so it does not use abstraction during the planning process as PLEXUS does. Once an operator is added into a plan, OPIE will only specialize it. OPIE begins with an abstract operator and specializes it as more constraints become apparent whereas PLEXUS begins with a concrete operator and then abstracts and specializes as necessary to cope with situation differences.

Abstraction in ALPINE

Knoblock's approach [Knoblock 1990] to abstraction differs from Anderson's in that it deletes literals from states as well as from operators. Knoblock calls this *reduced-model* abstraction. Knoblock's system ALPINE analyzes a domain in terms of operator dependencies and automatically partitions literals and operators into abstraction levels. Literals at the highest abstraction level can not be changed by literals at any lower level. Knoblock calls this property of abstraction hierarchies *ordered monotonicity.*

Ordered monotonicity is useful in hierarchical planning because it allows the planner (which is called PRODIGY/ALPINE) to constrain its search. If there is no solution to a problem at a given abstraction level then there is no reason to continue the refinement because no operators at lower levels will be able to establish the goal. Instead the planner backtracks to a higher level and selects a different abstract plan to refine.

ALPINE creates the abstraction hierarchy by analyzing interactions between literals. It first states constraints on which levels each literal can occupy. All effects of an operator must be at the same level and they must be at the same level or at a higher level than the

preconditions of that operator. ALPINE represents these constraints in a graph where each node is a literal and a directed arc represents the constraint on levels.

Next, ALPINE reduces the graph to a directed-acyclic graph (DAG) where the nodes of the DAG are the strongly-connected components of the original constraint graph. It topologically sorts the nodes of the DAG to find a partial ordering and then heuristically adds further constraints to totally order the nodes, which become the abstraction levels of the hierarchy.

Because ALPINE is used with reduced-model abstraction, the planner ignores all literals at abstraction levels below the current level, both in operators and in state descriptions.This distinguishes it from planners that use *relaxed-model* abstraction, where the planner ignores less critical preconditions at higher abstraction levels but never ignores literals in state descriptions. ABSTRIPS [Sacerdoti 1974] was one of the first planners to use relaxed-model abstraction which is basically a means of ordering goals [Lansky 1992]. Knoblock shows how reduced-model abstraction avoids the relaxed-model problem of inconsistent states.

Both ALPINE and OPIE analyze a domain and build an abstraction hierarchy. Although ALPINE can produce an abstraction hierarchy for a domain, some domains lead to trivial (one-level) hierarchies. ALPINE can also build problem-specific hierarchies that are more likely to produce useful hierarchies.

ALPINE and OPIE are designed for two different styles of problem solving. ALPINE produces abstractions of the entire problem space intended for length-first problem solving where a problem is solved completely at one level before the problem solver descends to the next lower level. OPIE is designed for use in least-commitment problem solving where operators are specialized as constraints become apparent. OPIE might specialize one operator completely while other operators remain abstract, but ALPINE will not.

There are similarities, however, because like OPIE, ALPINE deletes literals from operators as it moves up the hierarchy. A problem solver like PRODIGY/ALPINE could avoid commitment between operators that are indistinguishable at a given level, but this capability was not implemented in the problem solver described in Knoblock's thesis.

Localization

Localization [Lansky 1987, 1992] uses knowledge of a domain's structure to facilitate a divide-and-conquer approach to planning. The planner uses information about the structure of the domain to reason about interactions between subplans. If the domain is properly structured there will be few interactions between subplans and the complexity of the problem will be reduced.

Lansky [1992] compares localization to abstraction and concludes that localization is a more general principle. She argues that relaxed model abstraction is a way of ordering the selection of subgoals. Localization provides the means to reduce search by ordering subgoals, but it provides additional search-reduction techniques as well.

Localization reduces search by reducing the size of plans— the planner constructs smaller regional plans which are less complex than global plans— and by reducing the branching factor at each node in the search tree— the planner considers only operations in the current region, which is a subset of the entire domain.

Lansky's research on GEMPLAN and COLLAGE focuses on how planners can use a localized domain during planning. Her planners use a more general form of constraint satisfaction than other planners such as OPIE. Since Lansky shows that localization subsumes abstraction, it offers a wider set of possibilities.

However, Lansky compares localization to relaxed-model abstraction and doesn't relate it to the taxonomic abstraction used for OPIE's least-commitment strategy. GEMPLAN can represent decomposition requirements but doesn't represent specialization requirements as OPIE does, so use of specialization is one difference between the two planners. However, adding a specialization constraint to GEMPLAN would be relatively easy so a more important distinction is OPIE's ability to automatically form abstraction hierarchies and otherwise analyze a domain.

3.2 Evaluating Plans

We consider a change a plan change a plan repair if it allows an agent to achieve a goal by executing the changed plan. Thus goal achievement is an important form of plan evaluation. However, a planner cannot rely on goal achievement only. As Hanks [1990] points out, "one could imagine being able to satisfy one's goals, but at an unacceptably high cost."

To illustrate the difference between plans that all achieve the same goal, consider the four plans in Figure 3.7 that Abigail might use to get access to Basil's files.

Plan A	Abigail tries to log in by giving an arbitrary string as the password for Basil's user ID.
Plan B	Abigail tries to log in as Basil 10,000 times, each time giving a different string as the password.
Plan C	Abigail tells the system administrator that she will pay $10,000 for access to Basil's files.
Plan D	Abigail encrypts each entry in a large dictionary and compares the encrypted word with Basil's encrypted password.

Figure 3.7 Abigail's four plans.

Plan A is almost certain to fail and thus is very unreliable. Plan B, while more reliable, requires a prohibitive amount of time. Plan C is very expensive, perhaps prohibitively, for Abigail in monetary costs. Plan D is more reliable than Plan A (if Basil is a typical user, Plan D's chances of success are quite good), takes less time than Plan B and is less expensive than Plan C.

If we evaluate these plans only in terms of goal achievement (with access the only goal) they are equivalent. However, they are not equivalent when evaluated with respect to other criteria such as monetary cost and time. One possible way around this problem is to add additional goals, e.g. for saving money, but then we still must deal with the problems of conflicting goals and partial satisfaction of goals.

Instead of evaluating only in terms of goal achievement or adding secondary goals the planner can apply the concept of *expected utility*. Recently planning researchers have applied expected utility and other decision theory concepts to planning. Some planners reason about expected utility explicitly and in quantitative terms. Others (especially earlier planning researchers) reason about it implicitly or in qualitative terms.

Before looking at how different planners reason about the expected utility of plans, I give the definition of expected utility (as stated by Haddawy and Hanks [1990]):

$$EU(A) \equiv \sum_{s} P(s|A,S_0)U(s)$$

In this formula, A is an action (or a plan), S_0 is a given initial state, $P(s|A,S_0)$ is the probability that A will lead from S_0 to state s, and $U(s)$ is the utility of state s.

If a planner has numeric values for the probability and utility it can easily calculate the expected utility. However, obtaining numeric values is difficult and sometimes impossible. Some researchers have focused on methods for calculating the numeric values and some have taken a more qualitative and/or implicit approach to expected utility.

We can reason qualitatively about the expected utility of Abigail's four plans in the following way: Plan A has a low value of $P(s|A,S_0)$, Plans B and C have low values of $U(s)$ (although that depends on how the agent values time and money, of course), and Plan D has fairly high values for both $P(s|A,S_0)$ and $U(s)$.

By generalizing the preceding qualitative example, we see two ways to simplify calculation of expected utility. We can assume that both have the same probability of succeeding, and look for the plan that lead to a state with the highest utility value. Or, we can assume that we want to achieve a certain state (a goal) and look for the plan that is most likely to succeed.

In the following subsections I show how different planners use expected utility of plans, either implicitly, qualitatively or quantitatively.

Implicit Expected Utility

CHEF [Hammond 1989] uses two kinds of evaluation: evaluation of retrieved plans and evaluation of plan changes. To evaluate retrieved plans it uses a value hierarchy. The value hierarchy allows CHEF to compare the utility of various states. Rather than giving numeric utility values, it uses the relative positions of states in the value hierarchy to determine the relative utilities of the states. Thus it could use its goal hierarchy to decide whether to spend a lot of money as in Plan C or try to use Plan D. CHEF could not use its goal hierarchy to decide whether the systems administrator could be bribed (Plan C) because it reasons about only one agent.

In evaluating plan repair, CHEF uses various domain-dependent and domain-independent heuristics. For example, it prefers plans with fewer steps to plans with more steps, all else being equal. Thus if each log-in attempt were a step in a plan it would prefer Plan C (bribing the system administrator) over Plan B (trying to log in with 10,000 different passwords).

Although CHEF considers the fact that some plans will fail, it assumes that a given plan will always succeed or always fail. Thus it simplifies the expected utility by assuming that $P(s|A,S_0)$ will always be 0 or 1.

POLITICS [Carbonell 1981] reasons about utility of goals and about the relative likelihood of success for any plan to reach a goal. Unlike CHEF, which reasons about goals of a single agent, POLITICS reasons about the goals of multiple agents.

In the counterplanning model, one agent uses its knowledge of the relative importance (utility) of another agent's goals in order to influence that other agent's behavior. An agent also considers the importance of its own counterplanning goals to decide if the cost of a counterplan outweighs the benefit. Given Plan C, POLITICS would use its representation of relative goal values to determine whether Plan C would succeed, and if so, whether it would be acceptable to Abigail.

Like Carbonell, Wilensky [1983] describes the use of expected utility without actually mentioning the term or calculating numerical probabilities. Wilensky does talk about assigning numerical values to goals, but only discusses likelihood in general terms. Wilensky also discusses partial satisfaction of goals.

Hanks

Unlike earlier researchers in planning, Hanks looks at how numerical values for expected utility can be used in planning. He addressed probability of success for plans in his temporal projection work [Hanks 1990] and more recent work addresses utility models [Haddawy and Hanks 1990].

In particular, Haddawy and Hanks investigate the relationship of symbolic goals and numeric utility models and how to combine them. They state the importance of symbolic goals in planning and state how to create utility functions such that "choosing the plan that maximizes the probability of goal achievement maximizes goal utility."

Wellman

Wellman [1990] addresses the problem of tradeoff formulation. Tradeoff formulation involves reducing a problem to a choice among a small set of plans where the decision cannot be resolved by qualitative means. Wellman's system, SUDO-PLANNER uses qualitative probabilistic and preference information to find that small set of plans among the many plans possible in a combinatorially large set of plans. It compares plans and rejects those that can be ruled out by qualitative evaluation. Thus SUDO-PLANNER addresses the problem of qualitative evaluation of plans.

Wellman advocates the use of qualitative probabilistic networks because of the difficulty in constructing decision models. He argues that qualitative models are more robust and modular than quantitative models because they are not based on precise relationships that might change from one situation to the next. Instead they capture relevant causal information that does not vary.

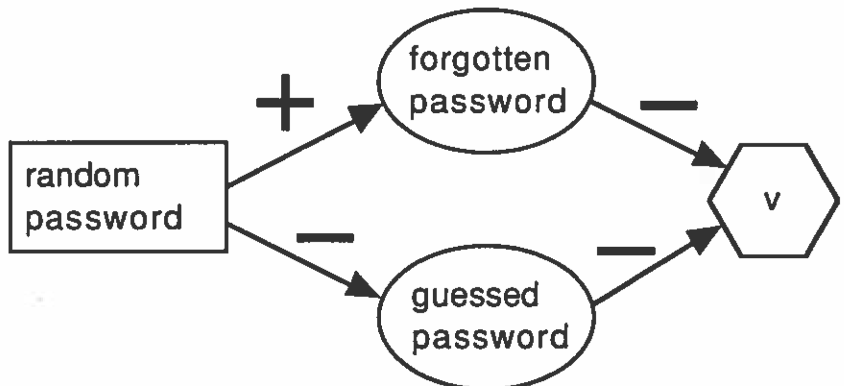

Figure 3.8: A qualitative probabilistic network (QPN).

As an example in the security domain, it would be difficult to construct a probabilistic model to calculate the probability that a user would forget his or her password. However, one can represent the causal link between random passwords and forgetting a password in a qualitative net as shown in Figure 3.8. In QPNs such as Figure 3.8, rectangular nodes represent decisions; the agent can determine their value. Ovals represent values that are not directly influenced by the agent and hexagonal nodes (v nodes) are utility or value nodes. Thus the overall goal is to increase the value of the v nodes.

The net in Figure 3.8 shows that users are more likely to forget random passwords and crackers are less likely to guess random passwords. This information is modular in the sense that it can be used in many different situations; a quantitative version of this same net would be valid only for a certain user in a certain situation.

Wellman defines a probabilistic base for QPNs and the transformations performed on them. The interpretation for the node from random password to forgotten password is that, all else being equal, a random password makes forgetting the password more likely. The 'all else being equal' condition is important because it allows the qualitative comparison to be stated in simpler terms.

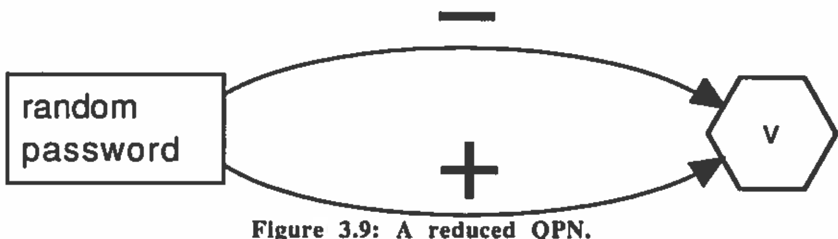

Figure 3.9: A reduced QPN.

Wellman describes how a qualitative probabilistic network such as the one shown in Figure 3.8 can be simplified by a set of transformations. For instance, sequential arcs such as the ones from random password to forgotten password and from forgotten password to v can be reduced by a transformation that replaces the arcs and the node with the product of the signs. Thus the top two arcs in Figure 3.8 can be reduced to the single negative arc in Figure 3.9 and the bottom two in Figure 3.8 can be reduced to the single positive arc in Figure 3.9.

In Wellman's terms, the situation in Figure 3.9 is a tradeoff because the decision cannot be qualitatively resolved. In other words, it is impossible to say whether it is better to have random passwords based only on the information in the QPN. Wellman's system is designed to analyze a situation and find the tradeoffs involved.

3.3 Contributions from Planning

I've described the following results from planning research:

Techniques for analyzing event dependencies allow a planner to determine how adding and deleting events will change a plan. All planners use some analysis of dependencies but repair and counterplanning results are especially useful for deciding whether a change will repair or break a plan.

Event abstraction hierarchies are important in planning, especially in adaptive planning and least-commitment planning. Some programs, such as OPIE and ALPINE, generate their

own hierarchies and others rely on previously generated hierarchies. Localization is a more general concept that subsumes abstraction and can be used to make a set of plans consistent.

Expected utility is a decision theory concept that has recently been applied to planning. It helps the designer compare plans based on their likelihood of success and their cost or benefit. Planning researchers have ways of reasoning about the likelihood and utility of plans. Some planners reason about expected utility implicitly and others explicitly, and some use numeric values and others use qualitative values.

So far I have only discussed how these techniques are used in planning problems. Although they can also be used in design evaluation with scenarios, some issues arise in design applications that do not arise in planning. In the next section I will discuss issues in using these techniques for design evaluation with scenarios.

4. APPLICATION TO DESIGN EVALUATION

In this part of the paper I will discuss how the planning results described in Section 3 can be applied to design. Since a planner changes only the plan but the designer changes the operator set, planners do not address all the issues that designers do. In this section I discuss some issues that emerge in using planning techniques in design evaluation.

4.1 Enabling and Disabling Scenarios in Design

When a designer eliminates a deficiency she is essentially fixing or breaking a plan and so can use plan analysis techniques to decide the best design change. To do so, she must map plan changes to design changes.

Figure 4.1 shows how plan changes map to design changes. If deleting an operator breaks a plan, then the designer can eliminate the corresponding deficiency by deleting the operator from the operator set and thus forcing its deletion from the plan. Similarly, if adding an operator fixes a plan then adding the operator to the operator set will eliminate the deficiency.

However, things are not always so simple. For example, suppose that Abigail finds out Basil's password because Basil told it to her. The designer cannot simply eliminate the operator tell password because it would be extremely difficult to prevent Basil from telling his password to anyone. Contrast that change with a change that prevents the system from displaying passwords on the screen. In the second case we are changing the functionality of the artifact, which is under the control of the designer. Eliminating the operator display password is a reasonable design change whereas eliminating tell password simply makes the model of the environment inaccurate. This example illustrates the problem of representing and reasoning about agents in the environment.

Agents in the Environment

As shown in the previous example, sometimes an operator cannot be eliminated from the operator set because the operator represents a feature of the artifact environment that is difficult or impossible to change. Deleting the operator would thus make the model of the environment inaccurate, rather than actually eliminating the deficiency. The designer cannot assume that agents will not use some functionality in the environment.

Anderson and Fickas [1989] describe a system that prevents the designer from making changes that make the model of the environment inaccurate by using two operator sets, one

to describe the functionality of the artifact and one to describe the functionality that is part of the environment. Their program could not change operators that were part of the environment.

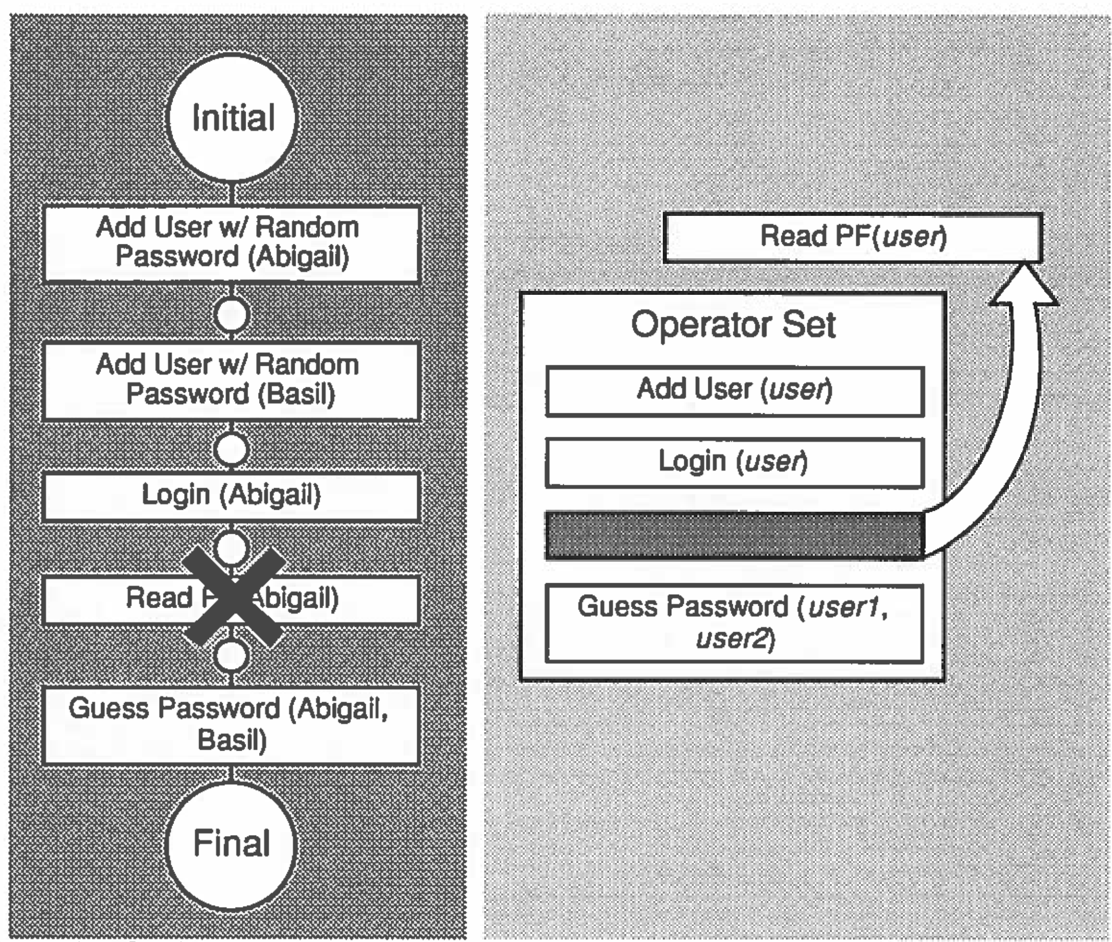

Figure 4.1: If disabling an operator in a plan will break the plan then deleting the operator from the operator set will disable the scenario.

Just as the designer cannot assume that agents will *not* use some operator in the environment, she cannot assume that an agent *will* use some functionality that is available in the artifact. For example, a designer might add functionality that helps users choose secure passwords, but that change alone will not disable scenarios where users choose easy-to-guess passwords because some users will not take advantage of the assistance. Thus the agents' choices determine whether a constraint will be satisfied or not.

Feather [1987] and Fickas and Helm [1991] have studied the problem of agents and how their choices relate to design constraints. They refer to this problem as one of *responsibility*. Their system finds ways of assigning responsibility for design goals and constraints to agents who are expected to limit their choices accordingly. If all agents fulfill their responsibilities the artifacts will satisfy its constraints.

The designer can influence the choices of a program but cannot directly change the behavior of an agent in the environment. However, she can indirectly influence it by offering incentives. For instance, a system could prevent users from logging in to the system if they have chosen a good password. This type of incentive requires knowledge of agents' goals.

POLITICS uses a model of agent goals in planning and counterplanning. Although constructing a detailed model of a person's goals is impossible, a simplified model of goal (such as the ones that POLITICS has for countries) is still useful. For instance, some computer security systems rely on the fact that people don't want to have privileges revoked, pay fines, spend time in jail, and so on.

POLITICS has a model of agent goals but it is basically a planning system and does not have the ability to decide how to change an operator set. A design evaluation program that addresses the problem of agents in the environment would need to be able to change operators in a way that appropriately influences agents' choices. Rather than deleting operators from the environment, or assuming that agents will use a certain operator in the artifact, the program would use a model of agent goals to change operators in a way that satisfies design constraints.

The designer can attempt to make changes that influence agents' choices but must realize that some agents will attempt to circumvent those changes. The designer must consider ways in which agents will adapt their plans in order to meet their goals. The next section describes the issue of adaptability and its ramifications in design evaluation.

Adaptability

Suppose that Abigail logs in to a system using her own log-in ID and then accesses the password file to get information that helps her guess Basil's password. One way of disabling this scenario, and thereby eliminating a deficiency, is to invalidate Abigail's log-in ID. However, the problem remains that other users could perform the same sequence, and Abigail could use other sources of information to guess Basil's password. In other words, the bad scenario can be adapted to design changes meant to disable it and thus become a new deficiency.

Given such a problem, the designer should make design changes as comprehensive as possible. Determining which changes are more comprehensive is the issue of adaptability. A designer should change the design so that bad scenarios are difficult or impossible and should also change the design so that good scenarios are robust.

Thus, when comparing two design changes, one criteria is how comprehensive the change is. The designer can use abstraction to evaluate changes in this way. Rather than break a concrete bad scenario, which might be easily adapted to become a new deficiency, the designer can look for changes that break an abstract one. Such a change would also break many adaptations of the original plan.

Figure 4.2 shows three scenarios where Abigail uses information from the password file to guess Basil's password. There are various design changes which will break one of these scenarios without breaking another.

For instance, changing the password selection action from user selection to random selection will disable the guess password action that uses a dictionary to find out the user's password. However, it still allows other ways of using the password file information to guess a password. The cracker can encrypt every password up to a certain length and compare the encrypted passwords to the password file ones. Abstraction hierarchies such as the one that OPIE uses allow a designer both to find a more abstract version of the scenario and to find a change that breaks that more abstract scenario.

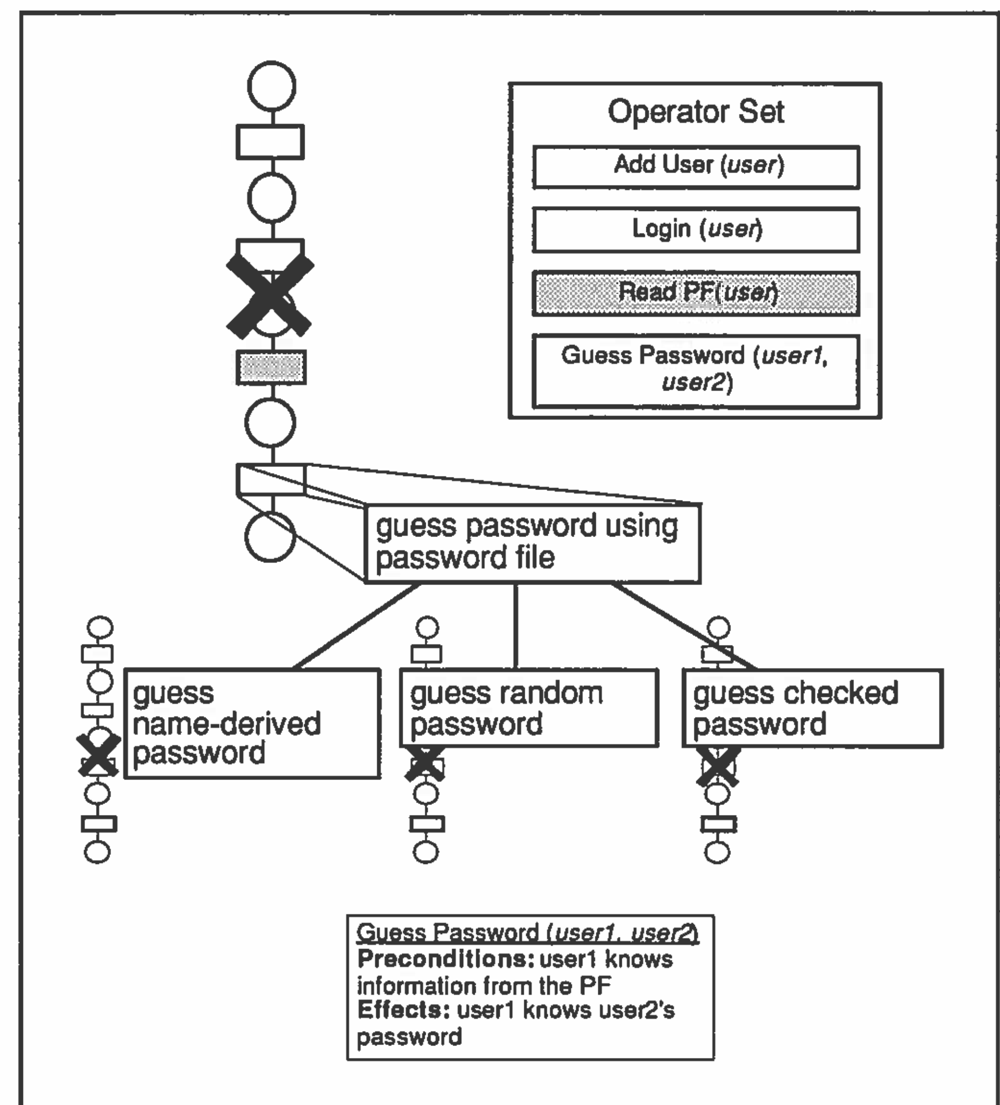

Figure 4.2: Disabling an abstract operator (by deleting the Read PF operator) disables all scenarios which use specialization of that abstract operator.

OPIE's taxonomic abstraction works well for addressing this issue. We can also consider how one might apply other abstraction techniques to this problem. For instance, one could imagine applying a form of relaxed-model abstraction by claiming that disabling a more critical precondition is a better fix for a bad plan than disabling another precondition. The problem with using criticality to address adaptability is that it might tell the designer something about how hard a precondition is to establish but it tells nothing about which alternate operators an agent might use.

Although OPIE's abstraction hierarchy appears promising, some issues in applying it to this new problem remain open. OPIE abstracts operators in storing macros but does not have the means of analyzing abstraction in the way required to address the issue of adaptability. It has no way of selecting among alternative abstractions and the research

results for OPIE do not include analysis of which scenarios will actually be disabled or enabled and which is the best abstraction to use.

Similarly, PLEXUS uses techniques for reasoning about which abstractions and specializations will enable a plan but it is concerned with only a single plan, not a set of scenarios, and it is concerned only with enabling plans, not disabling them. The designer can use abstraction, as used in PLEXUS and OPIE, to deal with the question of adaptability but new techniques will have to be developed because design problems are concerned with more than one scenario and with disabling plans as well as enabling them.

The designer anticipates and avoids future problems by looking at more abstract scenarios and then choosing the most comprehensive fix possible. She must also avoid past problems by ensuring that design changes don't undo early changes and cause old problems to reappear. This is the issue of regression.

Regression

For instance, suppose that the designer wants to eliminate a bad scenario and so changes the passwords in a system from user-chosen ones to random ones. A new deficiency arises wherein a user forgets his password and cannot access files so the designer must change the design again. One possible change would be to change the passwords to user-selected passwords, which are easier to remember. However, the designer would simply be undoing her earlier change. In this short example the regression is obvious but in a complete design the problem of determining whether a change undoes an earlier one is a difficult one.

By recording the design derivation the designer can avoid regression but doing so efficiently is difficult. Reason maintenance systems (RMS) [McDermott 1991] provide one way of determining the effects of a change in terms of previously discovered dependencies. When analyzing a deficiency, the designer can record the dependencies of that deficiency by using the reason maintenance system. When the designer is contemplating a design change later in the design, she can query the RMS to see if the proposed change enables known deficiencies.

An RMS could help with the problem of regression but as Lansky [1992] points out it requires expensive updating with each change. She notes that localization provides a less expensive, albeit heuristic, approach to representing "what effects what." Although localization offers a promising general approach to this problem, it does not include automated domain analysis as ALPINE does.

Automatic partitioning like ALPINE's is useful for the type of analysis required for determining the effect of operator changes on scenarios. Using an abstraction hierarchy like ALPINE's is another way that the designer could reason about which scenarios a design change might affect.

Suppose that an operator decides to delete an operator at level n in the hierarchy. There is no way that that operator could affect plans whose goals are at lower levels in the hierarchy because lower-level goals do not depend on higher-level operators. Furthermore, there is no way that adding an operator to a higher level could enable lower-level plans for the same reason. While it is fairly easy to tell if deleting an operator will disable a plan, it is more difficult to tell whether adding one will enable it so the abstraction hierarchy would be useful for the problem of regression.

The designer would not want to use exactly the same approach as ALPINE, however. Rather than creating a hierarchy she would use the DAG because it will indicate that a plan is independent of an operator change in more cases than the hierarchy would, as illustrated in Figure 4.3. In this figure, the DAG shows that plans in A are independent of changes to B and vice versa, whereas either of the possible hierarchies shows only one of these properties.

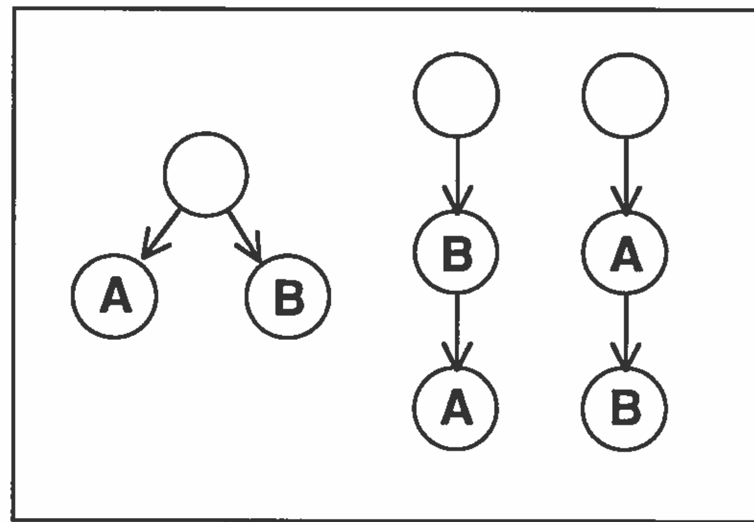

Figure 4.3: Independence of A and B relative to each other is shown in the DAG but not the hierarchies.

ALPINE provides useful techniques for the problem of regression but localization provides a more comprehensive framework and offers additional possibilities for partitioning operator sets. However, ALPINE relies on problem-specific abstraction hierarchies when domain hierarchies are trivial which is not an option in design. The design problem is essentially to create the domain (in ALPINE's terms) so it is not possible to reason about hierarchies only in terms of a single planning problem.

Knoblock describes the use of abstraction hierarchies in planning and proves properties of the hierarchies that are useful for planning hierarchies. In a design problem, instead of knowing the effects of adding an operator to a plan, the designer must know the effects of adding an operator to a level of the hierarchy. I have argued informally in this paper that the abstraction hierarchy will provide useful information about the effects of adding and deleting operators but further analysis and proofs must be done before hierarchies like ALPINE's can be used in scenario-based design evaluation.

4.2 Evaluating Scenarios in Design

It is important to know what prohibited and desired states are accessible in a design, but it is also important to know how easily those states can be reached. A scenario shows that a certain state can be reached but to know the significance of that scenario and a design change that enables or disables it we must determine the expected utility of that scenario.

Consider again Abigail's four plans for getting access to Basil's files, as shown in Figure 4.4. From Abigail's perspective, the expected utility of Plan D is highest. Thus, from the designer's point of view, a design change which eliminates Plan D but not Plan C is better than a design change that eliminates Plan C but not Plan D.

Plan A	Abigail tries to log in by giving an arbitrary string as the password for Basil's user ID.
Plan B	Abigail tries to log in as Basil 10,000 times, each time giving a different string as the password.
Plan C	Abigail tells the system administrator that she will pay $10,000 for access to Basil's files.
Plan D	Abigail encrypts each entry in a large dictionary and compares the encrypted word with Basil's encrypted password.

Figure 4.4 Abigail's four plans (again).

Thus we can see that the expected utility of plans is important in evaluating design changes, but there are some issues in applying expected utility of plans to design that don't arise in other applications of the concept.

Likelihood and Utility Values

Suppose a designer is contemplating the question of whether to allow users to choose their own passwords or be assigned random passwords by the system. On the one hand, user-chosen passwords are easier to guess, but on the other hand random passwords are difficult to remember.

Given such a choice, the designer must determine which of two scenarios is more important: the scenario where Abigail guesses Basil's user-chosen password, or the scenario where Basil forgets his random password and so cannot log in. Knowing the expected utility of the plans can help the designer choose between the alternatives, but it is often difficult to calculate absolute expected utilities. In general, the designer faces the problem of determining likelihood and utility values.

If the designer is comparing similar scenarios, qualitative or relative probabilities and utilities are easier to determine and might provide sufficient information for the decision. In planning problems this is often the case because the choice is not between two completely different plans but between two different steps of similar plans. However, in design problems, a designer might need to compare two scenarios that have few, if any, steps in common.

Furthermore, because of the subjective judgment required in design (and often in planning as well) we cannot expected an automated system to fully evaluate the alternatives, but we do expect it to provide as much information to the human designer as possible. Because of the difficulty of assigning numeric probability and utility values much of the evaluation must be based on qualitative reasoning about expected utility.

Multiple Perspectives: Who Gets the Check?

In discussing Abigail's four plans I noted that the designer would like to minimize the expected utility of Abigail's cracking plans. In other words, what's bad for Abigail's cracking plans is (usually) good for the system and thus for the system designer.

However, things are not always that simple. Suppose that the designer must choose between two design changes where one change makes things easier for the system administrator and another makes things easier for users. The designer must consider the utility of the system from the perspective of the various users and stakeholders in the system, and must somehow resolve the conflicts that arise.

Robinson [1990] has studied conflict detection and resolution and applied decision-theoretic and negotiation techniques to the problem. While I do not propose to extend his work, I do note the importance of multiple perspectives and of evaluating design changes in a way that facilitates interaction with a system that resolves conflicts.

Many planning systems work from the perspective of a single agent and thus do not represent the information that Robinson's system requires. It is not sufficient for a designer to calculate the expected utility of a plan, she must calculate the expected utility of the plan from the perspective of each important agent or stakeholder and then resolve conflicts with the help of a system like Robinson's.

Artifact Costs vs. Plan Costs

When a planner calculates the expected utility of a plan it can analyze the utility of the plan in terms of the cost in expended resources of the actions, and in terms of the benefits of reaching the goal. For instance, in gaining access to Basil's files Abigail can compute the time and money spent and compare it to whatever benefit she gets from access to secret information.

Now consider the changes that a designer might make to prevent users from accessing others' files without permission. Part of the utility of such a change accrues from changes in plan utility, i.e., some high-cost plans are disabled and some high-utility plans are still enabled. However, changes in the cost of the artifact affect the utility of the design change even if they don't affect the utility of the scenarios involved.

For instance, suppose that the designer decides to install a machine to check dynamic signatures before allowing users access to files. Such a change disables some illegitimate access scenarios and leaves most legitimate access scenarios enabled. The expected utility of these plans might look good compared to another design change, such as assigning random passwords to users. However, the designer must also take into account the high cost of the hardware that the system requires to test dynamic signatures.

Generally, in evaluating design changes the designer must reconcile changes in expected utility of plans and in costs and utility of the artifact itself.

5. RESEARCH PROPOSAL

I plan to focus on the issues of adaptability and regression. In order to apply the planning techniques described in this paper to these issues I will need to extend the work on abstraction and dependency analysis.

In particular I will analyze the use of dependency-DAGs (similar to ALPINE's abstraction hierarchies) in reasoning about regression. I will state as heuristics or prove properties of dependency DAGs that allow the designer to determine which scenarios might be enabled or disabled by a given design change.

I will also analyze the use of abstraction hierarchies, such as the ones that OPIE generates, in terms of using them to address the issue of adaptability. Rather than using them only to enable plans I will consider how they can be used to disable them, and rather than reasoning about a single plan I will consider how they can be used to reason about sets of plans.

Using techniques developed in my research together with those described in this paper I will create a program to evaluate design changes as described in Section 2.5. I will

evaluate the techniques by comparing the program's work to that of human designers by using the program to rationalize existing designs.

Although the research results will be domain-independent, evaluation by rationalization will require the program to work in a domain so that its evaluations can be compared with a person's evaluation. I plan to use the domain of computer security.

An advantage to using the domain of computer security is that it is a "real" domain. Computer security is a research domain in its own right and the question of how to design the best security system for a particular set of circumstances is still an open one. There are many examples of security design (or lack thereof) available and virtually every computer installation requires some sort of security analysis so rational reconstruction is a viable approach to evaluation.

6. SUMMARY

One way to help a designer evaluate design changes is to show her the consequences of design changes in terms of scenarios. A computer program can indicate which scenarios are enabled or disabled and whether the change violates or helps satisfy a constraint. Furthermore, a computer program can also help a designer decide which scenarios are more important by helping the designer determine the expected utility of scenarios.

I've discussed how planning and plan transformation relates to design and stated some planning results that apply to design evaluation with scenarios. These results are:

- Analysis of Event Dependencies
- Event Abstraction Hierarchies
- Expected Utility of Plans

After discussing relevant work in planning I examined issues that arise in applying planning results to design. These issues are:

- Agents in the Environment
- Adaptability
- Regression
- Relative Evaluation
- Multiple Perspectives
- Artifact Costs vs. Plan Costs

I plan to focus on the issues of adaptability and regression and will explore the use of dependency analysis and abstraction to evaluate design changes in terms of their generality and their interaction with earlier design changes.

REFERENCES

Alterman, R., "Adaptive Planning", *Cognitive Science* **12**, 393-421, 1988.

Anderson, J. S. and A. M. Farley, "Plan abstraction based on operator generalization," *Proceedings of AAAI-88*, St. Paul, 100-104, 1988.

Anderson, J. S. and A. M. Farley, "Incremental selection in plan composition," University of Oregon CS-TR-90-11, 1990.

Anderson, J. S. and S. Fickas, "A proposed perspective shift: viewing specification design as a planning problem." In *Proceedings of the 5th International Workshop on Software Specification and Design*, 1989. Also appears in Partridge (ed), *Artificial Intelligence and Software Engineering*, Ablex, 1991.

Benner, K. and L. Johnson, "The Use of Scenarios for the Development and Validation of Specifications," in *Proceedings of the Computers in Aerospace VII Conference*, Monterey, CA, 1989.

Carbonell, J. G., "Counterplanning: A Strategy-Based Model of Adversary Planning in Real-World Situations," *Artificial Intelligence* **16**, 295-329, 1981.

Feather, M. S., "Language Support for the Specification and Development of Composite Systems," *ACM Transactions on Programming Languages and Systems*, **9**(2), 198-234, 1987.

Fickas, S. and R. Helm, "Acting Responsibly: Reasoning about Agents in a Multi-agent System," CIS-TR-91-02, Computer and Information Science Department, University of Oregon, 1991.

Haddawy, P. and S. Hanks, "Issues in decision-theoretic planning: symbolic goals and numeric utilities," in *Proceedings of the 1990 DARPA Workshop on Innovative Approaches to Planning, Scheduling and Control*, P. 48-58, San Diego, November 1990.

Hammond, K., *Case-Based Planning*, Academic Press, 1989.

Hanks, S., "Practical Temporal Projection," in *Proceedings of AAAI-90*, Boston, 1990.

Kant, E. and D. Steier, "The Roles of Execution and Analysis in Algorithm Design," *IEEE Transactions on Software Engineering*, **11**(11), 1375-1386, 1985.

Kant, E., "Understanding and Automating Algorithm Design," *IEEE Transactions on Software Engineering*, **11**(11), 1361-1374, November, 1985.

Kaufman, L. D., S. M. Thebaut and M. F. Interrante, "System Modeling for Scenario-Based Requirements Engineering," SERC-TR-33-F, September 1989.

Knoblock, C. A., *Automatically Generating Abstractions for Problem Solving*, PhD Thesis, Carnegie Mellon University, CMU-CS-91-120, May, 1991.

Lansky, A., "A Representation of Parallel Activity Based on Events, Structure, and Causality," in Reasoning about Actions and Plans, M. Georgeff and A. Lansky (editors), Morgan Kaufmann, 123-160, 1987.

Lansky, A., "Localized Representation and Planning Methods for Parallel Domains," *Proceedings of AAAI-87*, Seattle, Washington, 1987.

Lansky, A., "Localized event-based reasoning for multiagent domains," *Computational Intelligence*, **4**, 319-340, 1988.

McDermott, D., "A general framework for reason maintenance," *Artificial Intelligence* **50**, 289-329, 1991.

Lee, S. and S. Sluizer, "An Executable Language for Modelling Simple Behavior," *IEEE Transactions on Software Engineering*, 17(6), 527-543, June 1991.

Robinson, W., "A multi-agent view of requirements," *Proceedings of the 12th International Conference on Software Engineering*, Nice, France, 1990.

Sacerdoti, E. D., "Planning in a hierarchy of abstraction spaces," *Artificial Intelligence* **5**, 115-135, 1974.

Wellman, M. P., *Formulation of Tradeoffs in Planning Under Uncertainty*, Morgan Kaufmann, San Mateo, California, 1990.

Wexelblat, A., "Report on Scenario Technology," MCC Technical Report Number STP-139-87, 1987.

Wilensky, R., *Planning and Understanding*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.