An Architecture for Analogy-based plan improvisation

by

Swaroop S. Vattam

(Under the direction of Donald Nute)

Abstract

The central theme of this research is simple: people are able to cope with complexities of the real world because of their ability to improvise, which makes them more flexible and adaptable. Most artificial intelligence systems are rigid, do not scale-up to challenges of the real world, and do not exhibit improvisational behavior. If we acknowledge the benefits of improvisation methods for people, then we can also acknowledge that similar benefits apply to artificial intelligence systems. Therefore, if we can build systems that can improvise we are better-off.

Based on this simple theme, we have proposed a novel architecture that combines a traditional planner with an analogy-based improvisation component in order to craft planning systems that respond intelligently in complex and little-known environments. CREAP (CREative Action Planner) is an intelligent agent embodying our architecture. Situated in a simulated environment of its own, CREAP is designed to plan and act in ways that achieve certain prespecified goals. Experiments conducted on CREAP show that our architecture is capable of producing improvisational behavior, thus lending flexibility to our agent in responding to less familiar situations.

Index words: Artificial intelligence, Planning, Improvisation, Analogical reasoning, Testbed systems

An Architecture for Analogy-based plan improvisation

by

Swaroop S. Vattam

B.E., The University of Mysore, 1997

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

2003

An Architecture for Analogy-based plan improvisation

by

Swaroop S. Vattam

DEDICATION

To Vivek,

who never ceases to be a

source of inspiration

even to this day

Acknowledgments

I cannot think of anyone more directly responsible for completion of this thesis than my advisor, Donald Nute. I cannot begin to express my gratitude for his decision to sponsor my thesis and for all the support he has given me. Professor Nute, thank you for helping me over so many hurdles which initially seemed like tall brick walls, and for your patience in listening to interminable examples of Dragon-Gold and Hag-Jane scenarios.

I am also thankful to Dr Don Potter for agreeing to guide me through my independent study on analogical reasoning, which set the foundation for this thesis. I would also like to thank Dr Beth Preston for agreeing to be on my thesis committee and for the wonderful seminars in philosophy of mind, which have had a significant part to play in conceptualization of this thesis.

Many members of the Artificial Intelligence Center at UGA have contributed greatly to my work. A special mention goes to B. C. Bridger, who taught me to program in V-World and allowed me to take a look at his code, which saved me a lot of time. I would also like to thank Sanjay Chellapilla and Fred Maier for listening to my ideas from time-to-time and giving me valuable feedback.

Above all, I owe an incredible debt to my parents, my wife Reena, and my friend Archan for all their love and constant support.

Table of Contents

List of Figures

Chapter 1

Introduction

People try to cope with planning in complex and little-known situations by constantly improvising in the face of failure or uncertainly. This improvisation is an emergent process guided by the possession of a vast collection of background knowledge. Similar benefits may be achieved in automated planning systems by extending their role to incorporate improvisation mechanisms. In this chapter we describe what it means to extend the capabilities of an automated planner, our approach to the improvisation problem, and the goals of our research.

When people plan in everyday life, like planning a trip, planning a business venture or cooking for that matter, they do not blindly assume that they will find a perfect plan at first shot, nor do they assume that their initial plan will necessarily lead them to their goal. Obstacles arise both in creating a plan to solve the problem and in executing the solution. But in situations where the external world does not meet the expectations of the plan, people often improvise on the earlier plan. In our view, plan improvisation entails making certain goal-directed, local adjustments (could be mundane or creative) to parts of the original failed plan, with the aim of taking the planner closer to its goal.

A Motivating example

A certain person makes a plan to remove the mineralization off of a water faucet in her kitchen. Her initial plan is to clean the faucet with a normal household cleaner. When she executes this plan, she finds that the household cleaner is ineffective. She tries to think of some other cleaner to do the job. She is reminded of a situation when her friend had used some vinegar to clean the chrome bumper of her automobile and had applied some car wax to prevent further build-up. She now has a new plan, this time using vinegar instead of the household cleaner. She finds that there was some improvement, but did not completely remove the deposit. She then decides that it is not a job for any solution-based cleaner and thinks of a totally new approach. She plans to take a piece of sandpaper and scrape the deposit off. When she looks for sandpaper, she realizes that she has none at home. She is not happy with the prospect of running to the hardware store to get the sandpaper. Suddenly she is reminded of the pumice stone, of the kind she uses on her skin around the knees and elbows to rub off the dry skin. She grabs the stone from her bathroom and scrapes the faucet with it. It worked like a charm this time around.

This example is illustrative of everyday occurrence that we can relate to, and seems to be a natural example of goal-directed behavior. It is not easy to miss out the pattern underlying this example: an agent makes a plan to accomplish a certain goal, but the plan fails. Under such circumstances the agent improvises on the original plan, relying on her background knowledge. There are two important points to note in the above example that are very important for this research.

1. People are not flawless planners, nor do they always possess complete and correct information about situations in which they set out to plan. They arrive at an initial plan, act on it, measure, and improvise wherever needed.

2. In order to apply a plan effectively and flexibly in the face of greater variability than can be completely anticipated, people possess and use a vast collection of general knowledge or background knowledge gathered from their previous-experiences.

Although planning as a high-level cognitive activity has been addressed sufficiently in the field of artificial intelligence, the type of improvisational behavior indicated in the aforementioned example has received little attention. Most research on planning in AI focuses on perfecting the techniques of planning. Few address the question *"in spite of all the advances, even the best of automated planners can fail. What then?"* This question brings out the need for failure-driven models.

People fail too, either due to unforeseen circumstances or due to miscalculations. But people are open to obstacles, and they are flexible enough to make changes, improvise and learn from failure. Therefore, even the most sophisticated of automated planners needs to recognize situations in which it fails, and adopt flexible strategies to deal with failure. This is very important for automated planners to succeed outside the *idealized Blocks World*.

Very generally speaking, dealing with plan failure has been addressed in AI as extensions to classical planning using two approaches: (1) *plan-execute-monitor-replan* strategy, and (2) *combine-planners-with-machine-learning* strategy. We shall not address dealing with plan failure in approaches other than classical planning because the scope of this thesis is restricted to demonstrating plan improvisation behavior within the classical planning framework.

In the *plan-execute-monitor-replan* approach, if the plan fails the planning agent replans. However the assumption here - the knowledge required for replanning is already a part of the planning knowledge-base - brings us back to the same question: what happens if it is not? So this approach is rigid too, and the system does not improve over time. Any addition to the knowledge comes about through knowledge-engineering.

In the *combine-planners-with-machine-learning* approach, the planning agent has an additional machine learning component that acquires planning knowledge autonomously using various techniques. This approach is one step better than the plan-execute-monitor-replan approach because it does not assume that knowledge required to repair a failed plan is already there. If it is there, fine. Else, there is hope that the learning component will acquire the required knowledge in the future.

Most of the machine learning techniques (with few exceptions) can be compared to "learning by transmission" or the "learning by acquisition" theories of learning. Learning in the context of machine learning is like going to a school or to an expert to learn. Learning is seen as an independent and purposeful activity in itself, the end product being new knowledge.

In this thesis we are interested in a different kind of behavior that we notice in people, *improvisation*. If things do not go according to their plan, people are quick to improvise; they find a quick fix, often inventing or creating in real time, without detailed preparation and by making use of resources at hand. As we progress it will become clear that what we are referring to improvisation is quite different from replanning in the plan-execute-monitor-replan approach.

Improvisation can mean many things in English language. When we say something is improvised we typically mean something that was done to face some unforeseen circumstance. We stick to a particular usage of the term "improvised", which is seen as a deviation from the original program that was to be followed. Accordingly, when we have a plan and the world behaves just as we expect, we have little need to improvise on that plan. On the other hand, when something goes wrong while planning or while following a plan we need to improvise.

When we say we need to improvise or not, it is important to clarify at what level of granularity of plans are we speaking of. Planning can happen at different levels. High-level planning involves general tasks like `(go to laundry, put clothes for wash, go to store, buy bananas, go to laundry, pick up clothes)`. Planning is also concerned with low-level activities like navigation and obstacle avoidance. When we say that there is little need to improvise, it is with respect to the high-level plan. For instance, if the plan is `(go to garage, get in car, drive to work)` and we require no improvisation, it does not mean that we are not making constant changes at lower levels of the plan execution like navigation, or responding to traffic lights, or avoiding obstacles. It means that the high-level plan need not change and the goal is still achieved. On the other hand, for the same plan if my car had died on a major street and I am called on to restart a stalled car in the middle of traffic, I may have to improvise, or make changes to the high-level plan itself.

How do we improvise? What is the basis of improvisation? It is difficult to arrive at a complete and adequate theory of improvisation. However, based on what we have discussed so far we can speculate that reasoning by analogy is an important mechanism for improvisation. When my car dies on the major street, I might be reminded of the occasional stalling that my lawnmower suffers because of the wires leading to its spark plugs getting loose, and I react to these situations by jiggling the wires. Because of this reminding I might do the same to the spark plugs in my car too. Or, I may be reminded of a situation when my car developed a flat tyre once and I had to summon a tow truck. This sort of reminding is crucial for improvisation, and an important source for improvisation is our *background knowledge* - earlier experiences or generalizations from them. But it is important to note that improvisation is always *goal-driven.*

The field of *analogical reasoning* has addressed this kind of behavior. The purpose of solving problems by analogy is to reuse past experience to guide the generation of solutions for new problems, thus avoiding a completely new search effort. Therefore we claim that analogical reasoning is an important mechanism involved in improvisation. This is a weak AI claim, however.

It is possible that people may use all kinds of different mechanisms to improvise, analogical reasoning being just one of them. However, as far as this thesis is concerned, we are interested in the kind of improvisation that analogical reasoning fosters. Henceforth, “improvisation” refers to a particular kind of improvisational behavior - one that occurs during a failure in planning or plan execution, and uses past experiences and analogical reminding to complete a plan or repair a failed plan.

Improvisation could also lead to learning. But the process of learning in improvisation is quite different from that of machine learning. If machine learning is like going to the school or an expert for learning, improvisation is like "everyday" learning, based on "our" background knowledge. In the machine learning type of learning approach, the knowledge which is learnt is the "end product" of an independent learning activity. In our approach, the knowledge which is learnt is the "by-product" of context-dependent and goal-driven improvisational activity.

Further, improvisation could lead to both mundane as well as creative solutions, which is rather difficult to explain using any logical deliberative processes. Some people have argued that the ability to perceive analogies as being crucial to exhibiting creative behavior. Computational models of analogical reasoning have addressed this issue. Therefore, by basing our plan improvisation theory on analogical reasoning we can explain how the nature of solutions that arise from improvisation can range from boringly obvious to remarkably creative.

The basis for improvisation is a vast collection of background knowledge. So, a good memory of past experiences and the ability to be reminded of some useful objects or situations is the crux to the improvisational process. Assuming that we endow our planning agent with each of these, we can describe the outline of our approach as follows: the planning agent is following a plan M and it fails in a situation S. It is reminded of a situation S' that is in "some way similar" to S (if at all such a situation exists in the memory). S' helps in reformulation of the failed situation S in terms of S'. Reformulating a problem is finding a new way to look at it, to describe it in different terms. This could lead to a solution in S based on what the agent did in S'. Therefore, we have a new solution for S that might repair the failed plan M to get M'. If M' succeeds, the inferred solution to S was correct and it can be stored in memory for future reference, thus learning a new piece of domain knowledge. In any future course of planning if situation S is encountered again, the planner has better chances of succeeding. Therefore when improvisation is successful, learning is seen as the by-product of improvisational activity rather than an end product of a decontextualized learning activity.

To summarize, this thesis focuses on laying a framework for building more flexible and adaptable planners through improvisation of targeted actions within the plan. Our approach involves execution monitoring of the plan as it unfolds, and targeted improvisation to resolve the apparent indeterminacies in the environment. Improvisation is the process of inventing, creating, or performing in real time, without detailed preparation and often by making use of resources at hand. The resulting "new work" of improvisation realizes a prescribed structure in an inventive form and also abets learning. Memory and analogical reasoning are fundamental to problem reformulation, which in turn is the basis for improvisation.

1.1 Improvisation in people

Just as humans are good everyday planners and problem-solvers, they are good improvisers too. The use of "everyday" has a secondary connotation here: that of less-than-perfect. It is a notion that people aim for a satisfactory solution that achieves their goal vis-*à*-vis the optimal solution sought by ideal planners and problem-solvers studied in AI. Herbert Simon ([32], p. 259) has referred to this process among people as *satisficing behavior*:

```
It appears probable that, however adaptive the behaviour of organisms
in learning and choice situations, this adaptiveness falls far short
of the ideal of "maximizing" postulated in economic theory. Evidently,
organisms adapt well enough to "satisfice"; they do not, in general,
"optimize."
```

In line with this notion of people as satisficing agents, we would like to attribute another quality to people, that of everyday improvisation.

Improvisation has been traditionally associated with certain human activities where the improvisation process is more conscious and pronounced, like cracking jokes or storytelling, theatrical and musical improvisation, etc. But it is also true that improvisation occurs in many other human activities like politics, measurement, cooking, architecture, games, and also in a variety of ad hoc day-to-day activities [2].

In line with this, we see improvisation as an "everyday" notion, a quick fix to failed situations rather than consistently optimizing algorithmic behavior, but always executed with the aim of achieving a certain goal. Further, we have narrowed the scope of improvisation to mean *plan improvisation.* Plan improvisation is need-based and goal-driven, and we always improvise on "something." This "something" in our case is a plan. With respect to planning, our claim is: *we may have a general to specific plan that we intend to follow to accomplish a course of activity (base plan), but we often stray from our initial plan as much as we follow it. We improvise on the initial plan, using the knowledge of the planning activity as well as our memory of the world that we inhabit. This ability lends us the flexibility needed in order to adapt to variations in the world around us, and to accomplish our goal despite unexpected obstacles that may arise.*

The desire to succeed in the face of failure or uncertainty could just be one of the many reasons why people improvise. People may also want to improvise because they are tired of routine. They may also improvise to challenge themselves to do better than they have in the past. The triggers to improvisational behavior could be varied and subject to psychological debate, which is beyond the scope of this work. For the purposes of this research the motivation for our agent to improvise comes from a planning failure. In other words, a failure in planning or its execution triggers improvisation.

So far we have examined the “why” of plan improvisation. We have claimed that people, in the course of planning an activity and executing it, sometimes encounter problems that cause their initial plans to become invalid. To recover from such situations people tend to improvise rather than wipe the slate clean and start off again. Most of the time, the improvisations are so impromptu, subtle and effortless that people fail to take notice of them. Now, let us examine the “how” of plan improvisation with the help of the following example.

Let us assume that I make a plan to build a model of the leaning tower of Pisa for a science project, which involves a step that requires me to sketch the blueprint on paper first. Let us further assume that when I get to this step I realize that my 2B pencil needs sharpening, but I don’t have access to a pencil sharpener. This unforeseen situation doesn’t force me to abandon my current plan, pursue an independent learning activity to update my domain knowledge, and start planning from first-principles once again. Instead I improvise on my plan, tweak it a little bit, make some local adjustments based on what my goal is and what I already know about the external world. In this case, I may include a step that uses the edge of a pair of scissors to scrape the wood from the lead. Or, I may use a pen as a satisficing solution. It could be any number of alternate plans, based on the person and what he already knows about the world, and also on what he is reminded of at that moment.

As can be seen from the above example, when things don’t go according to the plan, there is an extremely large number of alternatives that one can use to improvise. Most of the time, we improvise on our initial plans to a fairly slight degree (trying to deviate as little as possible and trying the obvious). That is because of the cognitive resources involved. Trying out more common solutions - that have either worked in the past or are known to have worked for others - takes little intellectual work (e.g. substituting scissors for a pencil sharpener, or a pen for a pencil are quite obvious and take little thought to recall). However, in more extreme situations (where common solutions fail) we may invest more time and cognitive resources delving into the working principles behind the activity, to perform the activity in more novel ways. In this case, the structure of wood and the locale might remind me of the cheese grater and how I use it to grate carrots. I might borrow the principle behind grating carrots and use it to scrape the wood off the lead. The more unusual the alternative, the more intellectual work it takes to think of.

Substitutions such as these and the ability to transfer knowledge from a similar situation in the past to the current situation are at the heart of improvisation. Improvisation requires the ability to integrate *knowledge about the current* and *knowledge from the past*. Knowledge about the current includes the current plan, the goal, the observations (like noticing that the sharpener is unavailable), and the causal course of events as the plan unfolded. Knowledge about the past includes previous planning or problem-solving episodes, lessons-learnt or generalizations gathered over many situations (like the car won't start unless there is fuel in the tank), reported experiences of other people, or knowledge contained in fictional accounts like stories, fables, etc. This method of problem-solving, inferring new facts about the current using knowledge from similar situations in the past is referred to as analogical reasoning, which will be discussed in detail in chapter 2.

To summarize, improvisation is an "everyday" notion and not a consistently optimizing algorithmic behavior. People improvise all the time by finding and learning new ways to achieve their goals when already established procedures do not work. On the one hand, some improvisations are so impromptu, subtle and effortless that people fail to take notice of them. On the other hand, some improvisations are novel and creative. Improvisation requires the ability to integrate knowledge about the current and knowledge from the past. An important source for improvisation in people could be either specific episodes from their past or general knowledge gathered over time. Therefore, analogy can be considered as an important mechanism for improvisation.

1.2 Improvisation in artificial intelligence systems

The central theme of this research is simple: people can cope with complexities of the real world because of their ability to improvise, which makes them more flexible and adaptable. Most artificial intelligence systems are rigid, do not scale-up to challenges of the real world, and do not exhibit improvisational behavior. If we acknowledge the benefits of improvisation methods for people, then we can also acknowledge that similar benefits apply to artificial intelligence systems. Therefore, if we can build systems that can improvise we are better-off.

As mentioned earlier, improvisational behavior in people is seen across a variety of activities. In this thesis, however, we are focusing on improvisation in artificial intelligence systems through the narrow window of planning. Therefore this research falls under the umbrella of artificial intelligence planning, which has been extensively researched. In order to lay the groundwork for this research we have to concern ourselves with related work in artificial intelligence planning, especially looking for approaches that address the issue of *building flexible and adaptable planners that can cope with real world complexities.* This research also treads on lesser-known waters of theory of improvisation. Let us first look at planning and related topics.

Artificial intelligence planning includes classical planning, memory based planning and reactive planning. Then there are also hybrid approaches that combine one or more of these approaches. Extensions to classical planning, which relax some of its unrealistic assumptions, have also led to the development of a host of superior planners. There have also been attempts to combine other AI paradigms with planning, like machine learning. As we can see, there have been numerous approaches and myriad attempts at building automated planners. However, most of the research in automated planning is focused on perfecting the techniques of planning, producing better quality plans and/or improving the efficiency of planners. Few approaches address the issue of making planners more autonomous and flexible, the issue in which we are interested in this thesis.

Machine learning techniques have been used extensively in the past to help planning systems. Manually encoding the knowledge required for the planners is a time consuming and tedious activity. Therefore, many approaches integrate machine learning techniques with planners to make them more flexible and autonomous. These systems are a step better than just planners because they acquire knowledge incrementally, and the system gets better over time. Planners that use machine learning come closest to our effort here.

Planning systems that incorporate machine learning techniques can be categorized into three categories. In the first category, also known as “speed-up” learning, machine learning techniques are used to acquire knowledge that improves the efficiency of a planner [7]. In the second category, machine learning has been used to acquire heuristics to guide the planner to produce plans of high quality ([29], [21]). In the third category, machine learning has been used to learn the domain knowledge for planning ([34]). We are not interested in the first two categories as we are not focusing on either the efficiency of the planner or the quality of the plans generated by the planner. The third category is more important to us because it concerns the knowledge of the planner, and the aim is to make the planner more autonomous and flexible. So let us concentrate on the third category.

This approach, where machine learning is used to learn domain knowledge, is driven by the assumption that a planning failure is caused when the expectations of the planner are not met, which in turn is due to the domain model (of the planner) not accurately reflecting the external world, i.e., the domain model is incomplete and/or incorrect, which needs to be rectified by the process of domain knowledge acquisition. In this approach, when the expectations and the observations diverge, learning is triggered. Modification of domain knowledge by means of learning is seen as a potential fix to the problem. The systems acquire new factual domain knowledge by one of the following mechanisms:

- **Direct specification** - requires a domain expert or knowledge engineer to formalize the knowledge as code or declarative statements in the system's representation language.
- **Programming by demonstration** - allows the system to inductively learn domain knowledge by watching a domain expert perform tasks.
- **Autonomous experimentation** - the system actively experiments in the external world to learn the preconditions and effects of actions.

If we observe closely, "direct specification" as well as "programming by demonstration" both use external sources of knowledge to learn. These methods are analogous to seeking the help of other experts to solve our problem. There is no doubt that we do that all the time. But that is not the kind of behavior in which we are interested in this research. Here, we are interested in solving our problem using what we already know, by using our background knowledge. "Autonomous experimentation," however, comes close. In "autonomous experimentation" the system learns by trying out various actions in the environment and by observing the effects of the actions. This is quite close to our approach. But the question is, can we consider systems conducting these autonomous experiments as exhibiting improvisational behavior? No, not in the sense that we have described improvisation. There is one crucial difference.

In learning by automated experimentation, the design of experiments to be conducted by the system is seen as a learning problem. Experiments are designed in a systematic algorithmic way, trying out various combinations of operators using search methods or other machine learning techniques like the version space algorithm. Our approach, too, encourages the system to improvise by trying out different solutions when an established procedure fails. But the knowledge for this improvisation is thought-out, goal-driven and there is a good basis for carrying out the experiment. The knowledge for this improvisation comes from analogical reasoning, from seeing the similarity of the current situation with some other situation, and by transfer of knowledge from one situation to the other. Therefore, our approach differs from all the above approaches because of a conceptual design of a planner that exhibits goal-driven improvisational behavior based on its background knowledge using analogical reasoning. That brings us to an area of research equally as important and vast as planning itself - analogical reasoning.

Just as approaches to planning are extensive, theories of analogy and analogical reasoning are numerous (e.g. [11], [12], [20], [17]). These theories approach analogy in the most general sense, describing how some process of mapping can infer relations between concepts (e.g. *herd : buffalo : : school : fish*). These theories have made great strides in accounting for the mapping process, but few systems have incorporated these theories into more general theories of planning and problem-solving. We will be discussing more about analogical reasoning in chapter 2 on background.

To summarize, there are many approaches to planning but few exhibit flexibility and adaptability. One particular approach, which integrates machine learning and planning, is particularly relevant to our research because it does not assume a complete domain model and uses learning techniques to acquire domain knowledge, thus improving over time. This approach comes close but is not quite the same as our approach because of our emphasis on goal-driven improvisation, based on background knowledge and using analogical reasoning. Our research lies at the juncture of two vast and interesting research areas: planning and analogical reasoning.

1.3 Requirements for plan improvisation

In order to develop an intelligent system that can plan and improvise, the following are some of the things that are required. However, these requirements are based solely on our approach.

First, we require an automated ***planner*** with certain capabilities. It must be capable of synthesizing a plan given a goal as the input. There are additional constraints on the planner. The planner must be able to:

- interleave planning and execution;
- carryout a sensory action to fill in the required information;
- generate some form of an interrupt when execution fails at a particular step; and
- change goals and beliefs about the world to alter the order of actions in the plan.

We have mentioned various kinds of planners above (and these are further discussed in chapter 2), but our approach will be demonstrated within the framework of a classical planner. It is worthwhile speculating about the applicability of our approach to other forms of planners, especially because it has the potential to work there too, but that is beyond the scope of this thesis. For now, let us limit ourselves to a classical planner or one of the many versions of classical planning.

Second, a ***planning knowledge-base*** is required. The planning knowledge-base consists of the knowledge required by the planner to operate. It has two types of knowledge: *domain model* and *operator model*. The *domain model* consists of a set of facts that are true about the domain stored as part of the planner's knowledge. If a fact is not represented explicitly as true, then it is assumed to be unknown. Unknown situations are handled with the help of sensing actions. The *operator model* consists of all the legitimate actions that a planner can perform in the domain. The operator model can be incomplete or incorrect. Either of these situations could potentially lead to a plan failure, which then triggers the improvisation process.

Third, we need an ***analogical reasoning component***. The analogical reasoner is the component that is used to retrieve knowledge which is the basis for improvisation. The analogical reasoning component works by comparing and finding similarities between two situations. In this case one of the situations would be the situation in which the planner failed. The analogical reasoner looks for situations *similar* to the first and retrieves it from the memory, which would be the second situation. The *similarity* could be at the mere surface level or at the structural level. The second situation, retrieved from memory, forms the basis for improvisation. The nature of improvisation (mundane or creative) all depends on this second situation and a solution derived from it. The more sophisticated the analogical reasoner, the more creative the improvisations could be. In this research we have used the *Sapper* model for the analogical reasoning component. Details of this model will be presented in chapter 2 on background.

Finally, ***memory*** is an important requirement. Memory here refers to a collection of various pieces of information related to the history of the planner. This could range anywhere from previous planning and problem-solving episodes to lessons-learnt and generalizations gathered over many situations (like car won't start unless there is fuel in the tank). It could also include observations and experiences of other systems codified in the planner's knowledge. However, this information has to be represented symbolically and in a certain format that is usable by both the planner and the analogical reasoner.

1.4 The CREAP system

We have implemented in ***CREAP*** (***CRE***ative ***A***ction ***P***lanner) a system that plans as well as improvises whenever the situation demands. This system serves both as a testbed for our theoretical ideas about modeling improvisation and as a demonstration of the benefits of incorporating improvisational behavior into planners. CREAP's domain of operation is a simulated world in V-World. V-World is a testbed system which allows one to create custom-made simulated environments and develop softbots like CREAP that are situated in them. CREAP's world is a fable-like environment that has various other characters and objects besides CREAP like the queen, the dragon, the sword, etc. It also allows us to define their behavior programmatically. CREAP has to not only survive in its world, but also interact with other entities and achieve the ultimate goal that has been specified in advance, like rescuing the princess from the dragon and taking her back to the throne. This is a complex goal to achieve and CREAP has to first plan its actions and then follow the plan.

To begin with, let us assume that CREAP's domain model as well as its operator model is sufficient to arrive at an initial plan. Once it has the initial plan, CREAP begins to execute the plan in its world, monitoring the execution as the plan unfolds. If all the expectations of the plan are met, i.e. the external environment behaves as predicted, then the plan goes through and CREAP achieves its goal. This episode is stored in CREAP's memory for future reference. Note the assumptions we made here: the domain model as well as the operator model are sufficient and the external environment behaves as predicted. If things are so straightforward most traditional planners would work, permitting CREAP to achieve its goal.

In order to demonstrate the benefits of our approach, we relax the assumptions that we made above. We start the entire planning exercise with an incomplete domain model and an incomplete operator model. The external world of CREAP can be programmatically controlled. Therefore we make it dynamic and partially accessible to CREAP. With these changes, one or more of the following situations could arise:

- CREAP could fail to synthesize an initial plan;
- CREAP could fail during the execution of a particular action because one or more preconditions of that action could not be satisfied; or
- CREAP could fail during the execution of a particular action because the actual effect of performing an action did not match with the intended effect.

These are some of the expectation failures that we were referring to. In such situations, the improvisation component of CREAP can suspend the planning task while it attempts to find a solution to the failed situation.

In cases where CREAP cannot synthesize the initial plan, the improvisation process tries to formulate a new action that can be used by CREAP's planner in order to come up with an initial plan. In cases where CREAP fails during the execution because of an unsatisfied precondition, the improvisation process tries to formulate an action that satisfies the precondition of the failed action. In cases where CREAP fails during the execution because of the effect, the improvisation process attempts to formulate a new action that has the same effect as the action that failed.

The improvisation process does not formulate these new actions out of thin air. It uses an analogical reasoning component to draw analogues from the past and uses the knowledge from the analogues to formulate a new action. It is not guaranteed that the improvisation process will be able to formulate an action in all situations. It is subject to finding the right analogue from the memory. Therefore a repository of situations, which we refer to as memory, is required for the improvisation process to work. CREAP's initial memory consists of knowledge-engineered situations that aid in improvisation. But CREAP can easily be made to store all the planning situations that it encounters, thus incrementally building its memory of past-experiences.

Even if the improvisation process succeeds in its task, the resultant plan is still only a proposed procedure. It may or may not work. CREAP can only know that it works after it executes this procedure and achieves the goal at the end of it. If the plan goes through and the goal is achieved, there is a new piece of information that is learnt and corresponding updates are made to the domain and operator models. Thus, learning is seen as a by-product of the improvisation activity and brings about incremental changes to CREAP's planning knowledge-base.

1.5 Goals of this research

We claim that the process of improvisation will improve the performance of an overall planning system in complex and realistic domains. In particular, the ultimate goals of this thesis are:

- to develop an approach to improvisation that could be implemented and tested on a computational level;
- to create a framework for constructing planners that could benefit from this computational theory of improvisation;
- to implement and test this novel framework by means of a softbot that inhabits a simulated environment that is dynamic and partially accessible; and
- to show that automated planners can function even with incomplete or incorrect domain and operator models and get better over time.

1.6 Contribution of this thesis

This thesis began with the argument that the existing artificial intelligence systems are rigid and do not scale-up to challenges of the real world. People, on the other hand, cope with complexities of the real world. We attributed this to people's ability to improvise, among other things. In this thesis we acknowledge that similar benefits apply to artificial intelligence systems if they have the capability to improvise. Then we defined what we mean by improvisation. We also hypothesized that background knowledge is the basis for improvisation. We further claimed that analogical reasoning plays a vital role in retrieval of relevant background knowledge, problem reformulation and knowledge transfer which are the three internal processes associated with improvisation. In due course we will be presenting the architecture of a planning system that illustrates all our claims. We will also be presenting an implementation of this architecture in the form of CREAP, an agent developed within the V-World framework. V-World [4] is a testbed that provides both an artificial world for the agent to inhabit and a platform with which to iteratively prototype the computational processes making up the agent itself. Based on these, the chief contributions of this thesis are:

- proposing a computational theory of improvisation based on analogy;
- developing an architecture around this theory to demonstrate improvisational behavior in planning systems; and
- implementing this architecture in the form of an agent that is able to plan flexibly in the face of incomplete knowledge as well as uncertainty in the external world.

Chapter 2

Background

Our research builds on two interesting research areas: artificial intelligence planning and analogical reasoning. In this chapter we describe the approaches underlying our work, and the similarities and differences with other research in related areas. Since this project was developed within the framework of V-World, a simulation system, we will discuss a bit of V-World too.

Our research on CREAP stands on two broad areas of research: *AI planning* and *analogical reasoning*. CREAP uses AI planning techniques to create and execute plans and refines this process through improvisation by analogical reasoning. In this chapter we give an overview of various AI planning approaches and describe the fundamental tenets of classical AI planning which is the basis for CREAP's planner.

The focus of our work is not the planning aspects of CREAP, per se, but the improvisational aspects of CREAP. Within the domain of AI planning, two topics are closely related to our research: (1) planning systems that extend plans at execution time (also referred to as interleaving planning and acting) and (2) planning systems that learn and improve over time using various machine learning techniques. We discuss each of these in detail and discuss in what ways it is either similar or dissimilar to our work.

Our main focus is to explain improvisational behavior using analogical reasoning. Therefore we will also give an overview of the different computational theories of analogy and discuss the Sapper model, which is the analogical reasoning system used in this research.

Finally, our architecture is realized through CREAP, the softbot, which is situated in a simulated world. Both CREAP and its world were developed within the framework of V-World, a testbed system that provides a platform for developing and testing such agents. Therefore, we also discuss V-world in this chapter.

2.1 Artificial Intelligence Planning

Planning has been viewed as an active subfield of AI for a long time, and the literature on planning is vast and varied. It is outside the scope of this project to chronicle a detailed history of AI planning, or even to touch upon all of the issues of interest to planning researchers. Several summaries of work in this field and some seminal papers can be found in [1]. In this section we will present the three main approaches that researchers in AI planning have pursued: classical planning, memory based planning and reactive planning. We will, however, stress classical planning because our approach presented in this thesis can be considered as an extension of the classical planning approach.

2.1.1 Classical AI planning

Early AI research in planning used simplified planning domains in order to directly address the issues that were considered to be the core of planning. The following simplifying assumptions were made:

- Static environment: The planning agent is the only active entity in the environment; there are no other entities or natural forces that can interfere with the state of the world either during the planning process or during execution.
- No uncertainty: The planning agent's actions will always succeed and have the intended effect every time.

- Complete and correct planning knowledge: The planning agent's planning knowledge, comprising of action descriptions, is correct and complete, describing all the consequences exactly.
- The planning agent is capable of performing only one action at a time.
- Planning time is not an issue (best for it to be minimized, but is not really an issue).
- Execution time is not an issue (best for it to be minimized, but is not really an issue).

We will call planning in the context of these assumptions "classical planning." People have also used other terms to mean the same thing like "traditional planning" and "static-world planning."

Planning proves to be a difficult task even when we make such unrealistic and simplifying assumptions. Much of the early work in classical planning was based on a formalization of action known as *situation calculus*. A "situation" encapsulates a particular "state of the world," and the actions defined in this notation can be regarded as state-transformation operators [33]. This view invites the development of planning systems that use state-space search in order to discover action sequences that achieve the intended goal.

After this came one of the most influential of all classical planning systems, *STRIPS* [33]. In STRIPS, the action representations were made less expressive. They were called *operators* and were composed of three lists of formulae: the *preconditions* - which must be true for the operator to be applicable, the *add-list* (also known as effects) - application of the operator would cause them to be true, and the *delete-list* - application of the operator would cause them to be false. Although the STRIPS approach was quite influential, it was not free from problems. The STRIPS approach was associated with three types of problems in action theory: the *frame problem* and the associated *precondition qualification problem* and the *ramification problem* [33]. The solutions to these problems are still subjects of considerable controversy. So, even with the simplifying assumptions in place, developing a general purpose planner is difficult.

There have been some significant advances to STRIPS-style planning systems in recent years, both in the design of planning algorithms and their supporting knowledge representation formalism. STRIPS represents plans as linear (totally-ordered) sequence of actions. But some researchers have argued that the use of non-linear (partially-ordered) representations can reduce the size of the search-space (for e.g. see NOAH [33]). This non-linear approach is an example of a *least-commitment search strategy.* Also recently, TWEAK [33] formalizes a generic, partial-order planning system.

The systems mentioned above are only a few prominent ones among a multitude of classical planning systems. Unfortunately, analytic work has shown that all these improvements have a limit. These planning algorithms are condemned to exponential worst-case complexity and even in knowledge-impoverished *Blocks World* domain these planning problems are generally NP-hard [33].

These problems led to researchers looking for alternate approaches. Upon analyzing, it was understood that the complexities of planning were largely due to the generality of the problems that the planning systems were expected to solve. To overcome this limitation, researchers in the applied AI community began to *develop domain-dependent planners.* These were systems custom-built for particular applications (e.g., for controlling a particular factory). Although some of these systems worked successfully in narrow domains, they do not address the fundamental issues still troubling the classical planning approach. There are two main troubling issues associated with planners.

- Even with the simplifying assumptions in place (the ones mentioned at the beginning of this section), how can we build efficient planners that produce optimal plans?

- These simplifying assumptions seem so unrealistic. Can we relax one or more of these assumptions and still build planners that produce near-optimal plans and are reasonably efficient? Optimal plans and total efficiency are a bit too much to expect when we relax the assumptions.

In the wake of these problems, some groups of researchers focused on extending classical planning approaches to make them more practical. Other groups of researchers began advancing fundamentally different approaches to planning like reactive planning and memory-based planning.

2.1.2 Practical planners: Extensions to classical planning

The simplifying assumptions of classical planning prevented it from being very useful in the real world and as such were not very practical. Therefore, researchers began to question the simplifying assumptions. They tried to extend work in classical planning in ways that could allow it to relax one or more of these infamous assumptions. These variations of classical planners are similar to our approach because we use a classical planner too, integrated with an analogical reasoning component, of course, to extend its capabilities to exhibit improvisational behavior. Therefore, we can consider our research to be one of the extensions of classical planning.

Planning and acting

In real world domains, planning agents have to deal with incomplete or incorrect planning knowledge. Incompleteness arises because the real world is not completely accessible. For example, when I'm planning to buy a few things in a supermarket, one of them being milk, I may not know where the milk is. Incorrectness arises because the real world does not necessarily match the agent's model of it. For example, I might have known the price of milk to be $2.00/gallon. But the price might have doubled overnight. Also, planning agents might have to deal with the dynamic nature of the world itself, i.e., there could be other entities or natural forces that can interfere with the state of the world. For example, on my way to the supermarket, my car might go over a pointed object leading to a flat tyre.

The bulk of the recent work in extending classical planning to handle these situations involves extending plans at execution time. These approaches call for some type of monitoring of plan execution, and this monitoring would keep track of the actual world state at all times and be able to respond accordingly. Planning agents that use a strict classical planner execute their plans "with eyes closed." This is a fragile strategy as many things can go wrong. By monitoring the plan execution as it unfolds, the planning agent can tell when something is wrong, i.e., when the expectations of the planner are not met. It can then do ***replanning*** to find a way to achieve its goals from the new situation. Although there have been many implementation-specific variations, this overall approach is known as *interleaving planning and execution* ([35], [36]).

Planning and acting vs. our approach

From what has been discussed so far about planning and acting, it seems that this approach is similar to what we are trying to do in this thesis. The crucial difference, however, is in what happens after the execution monitoring detects that something is wrong.

In the planning and acting approach, if during the execution of the plan the planning agent discovered a discrepancy between the assumed world state and the true state, it has to ***replan*** from scratch. The problem with this is approach is that it assumes the knowledge required for replanning is already there, as part of the planning knowledge-base. Replanning is nothing but planning again, but from a different set of initial conditions. The operators required for replanning need to be part of the planning knowledge-base. Replanning is again deliberative, theorem-prover like, and from first-principles.

In our approach to plan improvisation, we are looking at replanning from a totally different perspective. We are claiming that replanning is not necessarily logical, theorem prover-like, constantly optimizing algorithmic behavior. On the other hand, what we are referring to as improvisational behavior is replanning which is driven by analogical reminding rather than logical deliberation.

Further, improvisation could lead to both mundane as well as creative solutions, which is rather difficult to explain using traditional replanning. By basing our improvisation theory on analogical reasoning we can explain how the nature of solutions that arise from improvisation can range from boringly obvious to remarkably creative.

2.1.3 Machine learning and planning

Planning and acting is just one of the extensions of classical planning. Planning and acting is a step better than classical planners because it acknowledges that the real world can be quite complex and also that an agent's knowledge can be imperfect. However, in terms of improving over time, these systems are pretty stiff. Sure, they may have 10 operators today and a human could enter 10 more tomorrow, thus expanding its knowledge and allowing it to achieve more goals. But that is not the kind of improvement over time that we are referring to. We are referring to planning systems that inherently learn and improve, like people, systems that have autonomous learning capability.

Recently, many researchers have addressed this issue by asking the question "what are planning systems supposed to learn on their own?" Generally speaking, researchers have integrated machine learning into planning systems for three purposes, building planners that: (1) learn to improve their efficiency of planning, (2) learn to improve the quality of the plans that they produce, and (3) learn domain knowledge.

The more general-purpose the planners, the more inefficient they are. Approaches to improve planning efficiency using machine learning have concentrated on learning certain domain-specific control knowledge that helps the planner to efficiently synthesize plans. To be more specific, the learning component is looking to learn: (1) any local strategies for applying operators, (2) recurring sequences of operators, or (3) decisions for optimal and reliable selection of operators. Several machine learning techniques have been used in this framework like learning macro-operators ([7], [24]), learning control-rules ([25]), learning abstraction hierarchies, etc.

Some frameworks have also used machine learning techniques to acquire domain heuristics that help the planner to produce plans of better quality. This is a recent approach and relatively little work has been done ([29], [21]).

The other important application of machine learning to planning has been *learning planning knowledge*. Let us call this framework *"machine learning for planning knowledge (MLPK)"*. The goal of an MLPK type of application is to incrementally improve incomplete or inaccurate planning operators, or to learn planning operators from scratch. Several efforts have focused on such applications using inductive learning techniques ([14], [34]).

Although producing high-quality plans in efficient ways is an important concern, it is not the focus of this research. However, application of machine learning to learn domain knowledge for planners (MLPK approaches) is very much related to our research. Therefore we will delve into this in a little more detail.

Machine learning for acquiring planning knowledge (MLPK)

Research in the MLPK framework of incrementally learning planning knowledge using autonomous learning techniques is relevant to us because it relaxes an important knowledge related assumption of classical planning - the planner possess complete and correct planning knowledge. Here is a framework that is open to planning failure, which says that "the planner might not start out with a perfect knowledge-base (which means it is liable to failure), but will acquire more and more knowledge whenever there is an opportunity to do so, thus getting better over time."

In classical planning, the planning knowledge is engineered by the domain experts. Acquiring this knowledge from the experts is a rather difficult task. It is not humanly possible to anticipate all the situations that the planner will encounter and enter all the knowledge required to tackle these situations. Therefore, planning knowledge is frequently incomplete. Also, the external world changes because of its dynamic nature and hence some of the facts in the planning knowledge-base may no longer be valid or correct. This leads to the incorrectness problem. Therefore in the classical planning framework, some expert has to identify the discrepancies in the planning knowledge-base and correct them manually.

The primary aim of machine learning within the MLPK framework is to reduce this knowledge engineering bottleneck. Normally machine learning techniques have been used to acquire the following pieces of planning knowledge: (1) preconditions of operators, (2) post-conditions or effects of operators and (3) entirely new operators. The systems that learn planning knowledge through machine learning techniques adopt one or more of the following approaches.

- **Autonomous experimentation**: This technique is used for refining the domain description if the properties of the planning domain are not entirely specified ([5], [14]). The opportunity for using this technique arises when the execution monitoring process notices a difference between the domain description and the behavior of the real world. When multiple explanations for the observed divergence are consistent with the existing planning knowledge, experiments to discriminate among these explanations are generated. The experimentation process isolates the deficient operator and inserts the discriminant condition or unforeseen side-effect to avoid similar impasses in future planning. Thus, experimentation is demand-driven and exploits both the internal state of the planner and any external feedback received.

- **Programming by demonstration**: This is also referred to as learning by observation. This is a novel method for acquiring the knowledge for planning from observation of expert planners and from its own practice [34]. The observations of the planning agent consist of: (1) the sequence of actions being executed, (2) the state in which each action is executed, and (3) the state resulting from the application of each action. Planning operators are learned from these observations in an incremental fashion, using some form of inductive generalization process. In order to refine the newly learnt operators, to make them correct and complete, the system uses these operators to solve practice problems and analyzes the traces of the resulting solutions or failures.

- **Direct specification**: This form of knowledge acquisition is akin to expert and apprentice-like dialogue. There is nothing autonomous about this approach. Usually the system provides a graphical interface that is directly connected to the planning process to facilitate the direct specification process. This helps not only in the specification of the knowledge by the user but also helps in the extraction of the expert's advise for guiding the search for a solution to the current problem.

Machine learning for planning knowledge (MLPK) vs. our approach

So far we have mentioned that machine learning techniques have been used for speeding up the planning process, improving the quality of the plans produced and also to learn planning knowledge for incremental improvement. We also mentioned that we are primarily interested in combined planning and machine learning systems that focus on learning planning knowledge (i.e. MLPK approach). So, how do MLPK and our approach compare with each other?

Both the approaches are similar in their objectives: to build more flexible and adaptable planners in the face of an imperfect planning knowledge-base. However, there are some important differences between these approaches. Most of the machine learning techniques (with few exceptions) can be compared to the "learning by transmission" or the "learning by acquisition" theories of learning. Learning in the context of machine learning is like going to a school or to an expert to learn. Learning is seen as an independent and purposeful activity in itself, the end product being new knowledge.

Improvisation could also lead to learning. But the process of learning in improvisation is quite different from that in machine learning. If machine learning is like going to the school or an expert, improvisation is like "everyday" learning, based on "our" background knowledge. In a machine learning type of learning approach, the knowledge which is learnt is the "end product" of an independent learning activity. In our approach, the knowledge which is learnt is the "by-product" of context-dependent and goal-driven improvisational activity.

Recall that the MLPK approach can be further categorized into: MLPK by Direct specification, MLPK by Programming by demonstration, and MLPK by Autonomous experimentation. The following table 2.1 shows the comparison between

Table 2.1: Comparison of various approaches including ours

	Coupling	**Learning**	**Source of knowledge**	**Underlying process**
Direct specification	Loose	Offline	External	Manual
Programming by demonstration	Loose	Offline	External	Inductive generalization
Autonomous experimentation	Tight	Online	Learning algorithm	Explanation-based learning
Our approach	Tight	Online	Background knowledge	Analogical reminding

the various learning techniques and our learning by improvisation approach using the following criteria.

- Coupling - the degree of independence between the learning and the planning components.
- Learning - online refers to learning as part of the planning process, offline means learning is an independent activity.
- Source of knowledge - source of knowledge for learning.
- Underlying process - the process used for learning knowledge.

To summarize, our approach can be considered as an extension to classical planning. Among the other extensions to classical planning, systems that interleave planning and acting set the precedent for our research. But there is a fundamental difference in how we perceive *replanning* - as analogical and not logical. Our approach is related to another line of research in planning, one that combines machine learning and planning (MLPK approaches). The difference is in the "school-like" learning vis-*à*-vis "everyday" learning.

2.1.4 Fundamentally different approaches to planning

So far we have discussed classical planning approaches, extensions to classical planning including the "interleaving planning and acting" approach and also the "combined machine learning and planning" approach. We compared and contrasted these approaches to our approach. In this section we will briefly take a look at other approaches to planning that are fundamentally different from classical planning. The difference is stark, and goes to the heart of each approach's ideology. We will be discussing two such approaches: "memory-based approaches to planning" and "reactive planning"

At the current point in our research, we dare not make any claims about the applicability of our improvisation theory to these non- classical planning approaches. However, it is worthwhile examining them briefly because we see some potential use for our theory in these approaches. The hope is that this section could inspire someone to carry our theory to a whole new frontier.

Memory-based approaches to planning

The AI paradigm commonly associated with memory-based approaches is known as *Case-based Reasoning (CBR)*. CBR involves recalling and adapting previous experiences and making use of them in new situations [23]. Planners adopting CBR approach make extensive use of episodic memory to store and refer to the previous planning episodes and the explanations for their success or failure in situations where they were used. CBR planners recall the specific instances and alter them to fit the new situation.

The CBR approach claims that people apply knowledge about specific experiences rather than general task knowledge, simply because generalizations lose a great deal of information which is available in specific instances. This type of reasoning has been useful in many situations: law, for instance, mainly involves analysis and application of previous cases to the current one.

There have also been attempts to build planners based on the CBR paradigm. Chef [23] is a system that plans meals. Chef makes use of episodic memory to store previous attempts at making meals (cases). If it is asked to plan a new meal, it adapts old recipes to create new ones.

So memory-based approaches to planning departs from classical planning in every possible way, right from the way knowledge is represented (as cases - CBR; as logical formulae - classical planning) to the planning process (recall based on similarity and adaptation - CBR; theorem-proving - classical planning). While it is certain that CBR techniques will eventually form an important part of advanced planning systems, case-based retrieval does not by itself obviate the need for more traditional planning. Unless case-bases contain plans for every eventuality, systems will still require algorithms for plan-modification, and perhaps also for "from-scratch" planning.

Reactive planning

Another departure from classical planning, to cope with complexity of planning, is to do very little planning (in a classical sense). To reduce the need for planning, researchers thought of building systems with some sort of "situated intelligence" [33]. This approach is based upon the premise that it is often possible to generate reasonable behavioral sequences by applying certain control rules at each moment of the systems activity. Such rules may help in choosing the next action to perform without long-term planning. The advantages to this approach, as the researchers claim, is that it equally works well (in theory) in static as well as dynamic external worlds, although just how well is a subject of considerable controversy.

The rejection of the static world assumption has led to this vast body of research, which is usually referred to as "reactive planning." The term "reaction" generally refers to the behavior that arises in the absence of goal-driven deliberation, hence the phrase "reactive planning." Reactive planning focuses on problems that do not arise in static worlds like uncertainty, possibility of failure, the effect of other agents or unpredictable natural forces, etc.

Although reactive planning plays an important role in automated planning, especially in real-time planning, many have objected to the performance requirements to bring about "sufficient" reactivity. Building a reactive system is a very complex and time-consuming activity owing to the requirement of pre-coding all of the behavior of the system.

2.1.5 Summary of AI planning

To summarize, our approach can be considered as an extension to the classical planning approach. Among the various extensions to classical planning, planning systems that interleave planning and acting set the precedent for our research. The defining characteristic of these systems is the *plan-execute-monitor-replan* cycle. Our approach, however, follows the *plan-execute-monitor-improvise* cycle. The crucial difference between replanning and improvisation is the following: replanning is again a theorem prover-like planning, but with a different set of initial conditions and assumes that knowledge required for replanning is already part of the knowledge-base. Replanning cannot account for the creative inventiveness that people come with to find alternate solutions. Improvisation, on the other hand, is based on analogical reminding.

Our approach is related to another line of research, one that combines machine learning and planning (MLPK approaches). Table 2.1 has already presented a detailed comparison between MLPK approaches and our approach. If machine learning approaches are like going to school or to an expert to learn, improvisation is like "everyday" learning, based on "our" background knowledge. In machine learning what is learnt is an end product of an independent learning activity. In our approach, it is the by-product of context-dependent and goal-driven improvisational activity.

The bottom-line is this: our approach, with background knowledge as the basis, uses analogical reminding to tackle a failed plan execution situation. This is not replanning (in the classical sense), nor is it the same as traditional machine learning. The novelty in this approach lies in its ability to explain how people sometimes come up with a range of solutions from boringly obvious to excitingly creative, just with the help of this one theory of analogy combined with our usual planners, which we have been studying for some time now.

2.2 Computational models of analogical reasoning

We are basing our theory of plan improvisation on analogical reasoning. This section looks at research on analogical reasoning. The ability of humans to perceive analogies has caught the attention of researchers in diverse fields, from philosophy and psychology to computer science. In this thesis we are interested in computational models of analogical reasoning. We will be discussing some prominent ones, and presenting the Sapper system, which is the analogical reasoning system that we have used in this research.

Link between analogical reasoning and model of plan improvisation

Very generally speaking, "analogical reasoning" or "analogy making" or simply referred to as "analogy" is the ability to think about relational patterns. For example, consider the following alphabetical patterns: $AABB$, $IJKL$, $IIJJ$, $YYXX$, $AABB$.

We can readily note that the pattern $AABB$ is the "same" as $AABB$, most similar to $IIJJ$, and more similar to $YYXX$ than $IJKL$. But where is this "similarity" that connects $AABB$ to $YYXX$ and not $AABB$ to $IJKL$? It does not reside in the physical forms of the letters, which do not overlap at all. Rather, it resides in the "relation" between the alphabetical characters. Therefore in very general terms, analogical reasoning is the ability to perceive and explicate such relational patterns among entities.

To be more specific, in analogy a given situation is understood by comparison with another similar situation. Analogy may be used to guide reasoning, to generate conjectures about an unfamiliar domain, or to generalize several experiences into an abstract schema [9].

In this research we are attempting to build more flexible automated planning systems that improvise. When the world behaves just as the planner expects, there is little need for improvisation. It is only when the execution of the plan fails that the planning system needs to improvise. We have already claimed that an improvisation process based on logical deliberation is not the right direction because the same issues of "where does the knowledge for logical inference come from?" and "can solutions generated from logical deliberation ever be inventive?" arise. Based on this and drawing inspiration from how people improvise when their plan fails, we are suggesting a model of plan improvisation based on analogical reasoning.

In our model of plan improvisation, analogical reasoning is used to generate conjectures about the situation in which the plan failed (unfamiliar domain) by comparison with other situations the planning system has seen in the past (familiar domains). These conjectures help in finding a potential solution for the failed situation. Incorporating this new solution into the original plan by making certain local adjustments to it, the planning system can complete the execution of the plan. Thus analogical reasoning is an important part of the plan improvisation process.

A brief history of analogy research

The roots of the research in analogy can be traced back to the elusive linguistic phenomenon called "metaphor." It is said that the ability to perceive analogy is at the heart of understanding metaphors in linguistic expressions. Metaphors have been in use since time immemorial. It is said that two thousand years of philosophical inquiry into metaphor has led to the Literalist and Figuralist manifesto [39], which is how long since man has been thinking about mental processes that constitute analogy. We will, however, begin with the modern view of analogy and especially from the AI point of view.

Modern views of analogy can be traced to the works of the philosopher Mary Hesse [19], who argued that analogies are powerful sources in scientific discovery and conceptual change. Later on for some time, most research on analogy, both in AI [19] and psychology focused on four-term analogy problems of the sort used in intelligence tests *(cat : kitten : : dog : ?)*, rather than richer analogies used in everyday life.

At about 1980, several researchers in AI began to take a broader view of the analogy and began to grapple with the use of complex analogies in reasoning and learning ([37], [31], [17]). This exploration led to a more general focus on the role of experience in reasoning and the relationship between reasoning, learning and memory. Dynamic memory theory [31] gave rise to an approach termed "case-based reasoning" in AI. In contrast to rule-based approaches to reasoning (the approach which was dominant at that time), case-based reasoning emphasizes the usefulness of retrieving and adapting cases (or analogues) stored in long-term memory when deriving solutions to novel problems.

In psychology, Gentner ([11], [12]) began working on mental models and analogy in science. She proposed that in analogy, the key similarities lie in the relationships (e.g., the flow of electrons in an electrical circuit is analogically similar to the flow of water in streams) rather than the features of individual objects. (For e.g., the electrons do not resemble water in terms of its features.) She also noted that analogical similarities often depend on higher-order relations - relations between relations. Based on this, she proposed a view that analogy is the process of finding a structural alignment, or mapping, between domains. Gentner and her colleagues carried out empirical studies to provide evidence for her theory, "the structure-mapping theory." The structure-mapping theory was eventually implemented on the computer to do

- analogical map and inferencing (the SME program [9]), and
- analogical retrieval (the MAC/FAC program [10])

These programs have been extended and applied to many domains.

Meanwhile, Holyoak and others were also investigating the role of analogy in complex cognitive tasks like problem solving. This led to the strong concern for the role of pragmatics in analogy - how current goals and context guide the interpretation of an analogy. Gick and Holyoak [13] provided evidence that analogy can provide the seed for forming new relational categories, by abstracting relational correspondences between examples for a class of problems. Holyoak and Thagard [20] developed a multi-constraint approach to analogy in which similarity, structural parallelism and pragmatics interact to produce an interpretation. They developed computational implementations in the form of ACME [20].

Over this period Douglas Hofstadter and his research group, since 1983, had been investigating the claim that the same mental mechanisms underlie seemingly disparate mental activities such as pattern recognition, analogies, counterfactuals, humor, translation between languages, etc. One of their conclusions was that *analogy making* lies at the core of understanding, and that *analogy making* itself is a process of high-level perception. Here, high-level perception refers to the recognition of objects, situations, or events at various levels of abstraction, much higher than that of sensations perceived through our senses [26]. They developed a computer program called Copycat to substantiate their claim.

Since the 1980s, the efforts of many AI researchers and psychologists have contributed to an emerging consensus (more or less) on many issues relating to analogy. The process of analogical thinking can be decomposed into several basic constituents. Typically, given a situation to be understood in terms of what we know (target domain) this is what happens.

- One or more analogues in long-term memory are accessed.
- An attempt is made to map a familiar analogue to the target domain.
- The resulting mapping allows analogical inferences to be made about the target domain, thus creating new knowledge to fill gaps in understanding.
- These inferences might have to be evaluated and adapted to fit the idiosyncrasies of the target domain.
- Finally, in the aftermath of reasoning, learning can result by the addition of new instances (new experiences) to memories.

Most of the current computational models of analogy deal with some subset of these basic component processes. In various ways and with differing emphases, all current models use underlying structural information about the sources and the target domains to derive analogies. (The emphasis is on structural or syntactic features, as opposed to semantic features.)

Overview of various computational models of analogy

A fair amount of work has been done in artificial intelligence and cognitive science on constructing computational models of analogy-making. Most of the models also agree on the following.

- Analogy making involves two situations: the *source* and the *target* situations/problems. For example, when a child concludes that "the tree is a bird's backyard," the child is looking for something in the world of the bird that corresponds to something in his familiar world. The child's everyday world is the source situation - a known domain that the child already understands in terms of familiar patterns, such as people sitting on chairs and houses that open onto the backyards. The bird's world is the target situation - a relatively unfamiliar domain that the child is trying to understand in terms of the source situation.
- Establishing an analogy involves making a mental leap, exploring connections between two very different domains (source and the target domains).

Analogical thinking in not "logical" in the sense of a logical deduction - in the above example of "the tree is a bird's backyard," there is no reason why birds and people should necessarily organize their habitats in comparable ways, nor can the child (or anybody else for that matter) prove such a conclusion. Yet, it is certainly not haphazard. We do see a similarity pattern between the two situations.

Most of the computational models concentrate on how a mapping can be established from a source problem, whose solution is known, to a target problem, whose solution is desired, with some kind of representation of the various objects, descriptions, and relations in the source and the target problem. These representations are provided to the reasoner (the program) at the outset. For example, from the first-order representation of the two situations in figure 2.1, SME (a famous analogy program) concludes that the "Solar system" and "Rutherford atom model" are analogous, based on the structural similarity of the two situations alone.

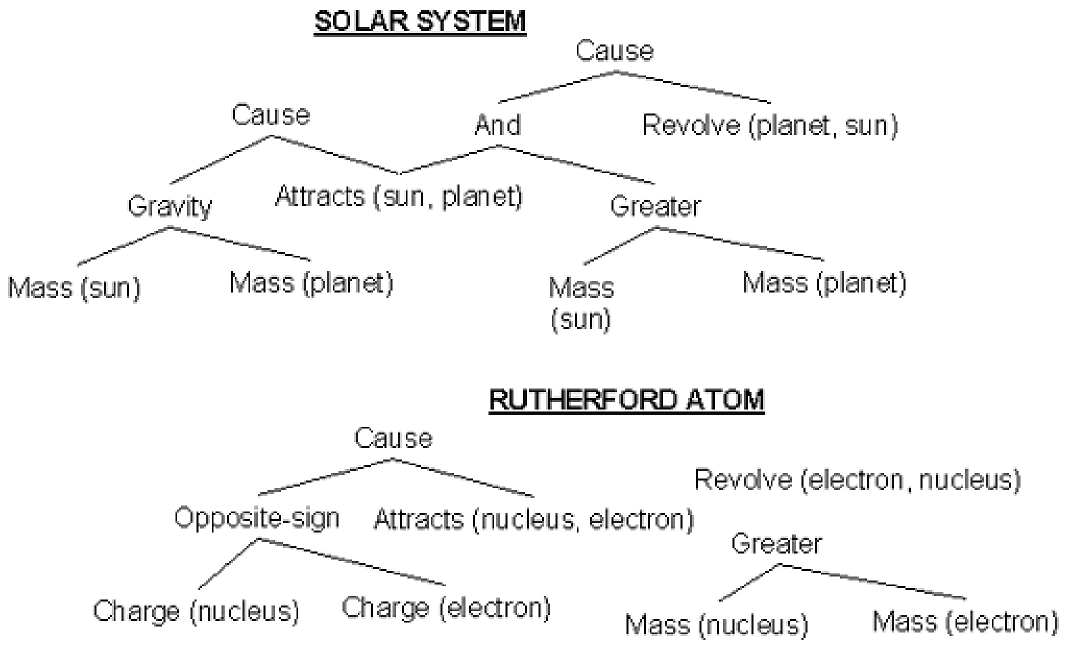


Figure 2.1: first-order representation of solar system and Rutherford atom models

SME and other similar programs expect the situations to be represented in a certain form (first-order logic mostly) and given to them at the outset. The majority of these models, however, do not focus on the *construction* of these representations for the source and target situations, and how this *construction process* interacts with the *mapping process*. This important point differentiates models like SME from programs like Copycat, to be discussed later. In what follows, rather than giving a broad survey of computational models of analogy making, we will discuss three important projects, chosen for their prominence in AI. This leaves out a good number of other models, described by Hall [16] and Kedar-Cabelli [22].

Structure mapping engine (SME)

Dedre Gentner's research is perhaps the best-known work in cognitive science on analogy. She formulated a theory of analogical mapping, called the "structure-mapping theory" [12], and she and her colleagues have constructed a computer model of this theory: the Structre-Mapping Engine, or SME [9]. The structure-mapping theory describes how mapping is carried out from a source situation to a less familiar target situation. The theory gives two principles for analogical mapping:

- relations between objects rather than attributes of objects are mapped from source to target situations.
- relations that are part of a coherent interconnected system are preferentially mapped over relatively isolated relations (the "systematicity" principle).

In effect, Gentner's definition of analogy presupposes these mapping principles. According to her, there are different kinds of comparisons: an "analogy" is a kind of comparison in which only relations are mapped, whereas a comparison in which both attributes and relations are mapped is a "literal similarity" rather than an analogy.

One of Gentner's examples of analogy is illustrated in the following figure 2.2 (taken from [9], p 5).

Consider the idea "heat flow is like water flow." This example has two situations: water-flow situation (source) and heat-flow (target). The representations of the two situations are as shown in Figure 2.2. The idea is that the causal-relation tree on the left is a systematic structure and should thus be mapped to the heat-flow situation, whereas other facts like "the diameter of the beaker is greater than the diameter of the vial," "water is a liquid," "water has flat top," etc. are less relevant and should

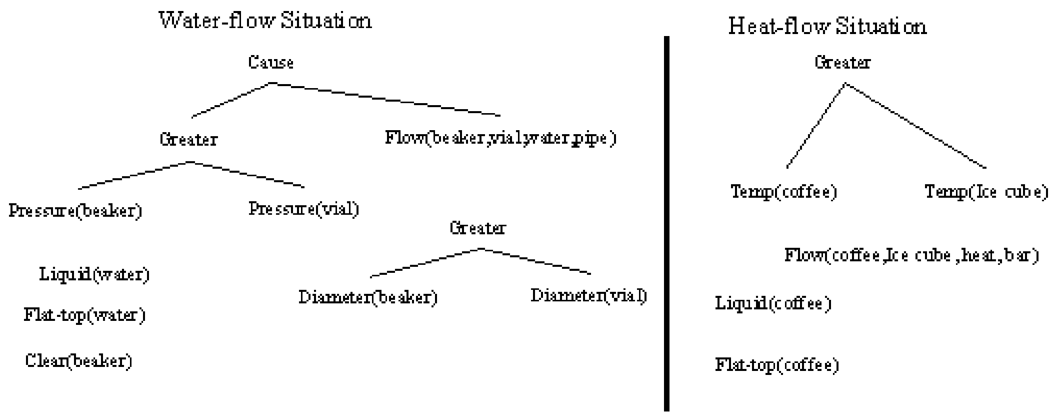


Figure 2.2: Water-flow and Heat-flow situations

be ignored. Ideally, mappings should be made between pressure and temperature, coffee and beaker, vial and ice cube, water and heat, pipe and bar, etc. Once these mappings are made a hypothesis about the cause of the heat flow can be made by analogy with the causal structure in the water flow situation on the left. Gentner claims that if people recognize that this causal structure is the deepest and the most interconnected system for this analogy, then they will favor it for the mapping.

Gentner gives the following criteria for judging the quality of an analogy:

- **Clarity** - a measure of how clear it is which things map onto which other things
- **Richness** - a measure of how many things in the source are mapped to the target
- **Abstractness** - a measure of how abstract the things mapped are, where the degree of "abstractness" of an attribute or a relation is its "order." Attributes

(e.g., “flat-top” in the example above) are of lowest order because they are one-arity, first order predicates, relations whose arguments are objects or attributes (e.g., “flow”) are of higher order (two/more arity first order predicates), and relations whose arguments are relations themselves (e.g., “cause”) are of even higher-order (second/higher order predicates).

- **Systematicity** - the degree to which the things mapped belong to a coherent interconnected system.

The computer model of this theory (SME) takes a predicate-logic representation of the two situations, such as the representation given in the figure 2.2, makes mappings between objects, between attributes, and between relations in the two situations, and then makes inferences from this mapping (such as “the greater temperature of the coffee causes heat to flow from the coffee to the ice cube”). The only knowledge this program has about the two situations is their syntactic structures (for e.g., the tree structure given above for the two situations); it has no knowledge of any kind of semantic similarity between various descriptions and relations in the two situations. All processing is based on syntactic structural features of the two given representations.

SME first uses a set of “match rules” (provided to the program ahead of time) to make all “possible” pairings between objects (e.g., water and heat) and between relations (e.g., flow in the case of water and flow in the case of heat). Examples of such rules: “if two relations have the same name, then pair them,” “if two objects play the same role in two already paired relations (i.e., are arguments in the same position), then pair them,” and “pair any two functional predicates” (e.g., pressure and temperature). SME then gives a score to each of these pairings, based on factors such as: Do the two things paired have the same name? What kind of things are they (objects, relations or functional predicates, etc)? Are they part of systematic structures? The kinds of pairings allowed and the scores given to them depend on the set of match rules to the program; different sets can be supplied.

Once all possible pairings have been made, the program makes all possible sets of consistent combinations of these pairings, making each set (or "global match") as large as possible. "Consistency" here means that each element can match only one other element, and a pair (e.g., pressure and temperature) is allowed to be in the global match only if all the arguments of each element are also paired up in the global match. For example, if temperature and pressure are paired, the beaker and coffee must be paired as well. Consistency ensures clarity of the analogy, and the fact that the sets are maximal shows a preference for richness. After all possible global matches have been formed, each is given a score based on the individual pairings it is made up of, the inferences it suggests, and its degree of systematicity. Gentner and her colleagues have compared the relative scores assigned by the program with the scores people gave to various analogies.

The Copycat architecture

Copycat is a computer program designed to be able to make analogies, and to do so in a psychologically realistic way. Copycat possesses neither a symbolic nor a connectionist architecture; rather the program has a novel type of architecture situated somewhere between these two extremes. It is an emergent architecture, in the sense that the program's overall behavior emerges as a statistical consequence of myriad small computational actions. The *"concepts"* that it uses to derive analogies are implemented as distributed, probabilistic entities in a network. This use of parallel and stochastic mechanisms makes Copycat somewhat similar in spirit to connectionist systems. However, as will be seen, there are important differences, which places Copycat in the middle ground in cognitive modeling, as a hybrid of symbolic and connectionist architectures.

The domain in which Copycat discovers analogies is very small. This idealized microworld, developed by Hofstadter, consists of 26 letters of the English alphabet as the basic objects. Analogy problems are constructed out of strings of letters, as in the following:

```
abc => abd
ijk => ?
```

Although a casual glance at the Copycat project might give the impression that it was specifically designed to handle analogies in a particular tiny domain, Hofstadter and his colleagues [17] claim that this is a serious misconception and that it was designed with an eye to great generality. Hofstadter claims that

... the Copycat project is not about simulating analogy-making per se but about simulating the very crux of human cognition: fluid concepts. The reason the project focuses upon analogy-making is that analogy-making is perhaps the quintessential mental activity where fluidity of concepts is called for, and the reason the project restricts its modeling of analogy-making to a specific and very small domain is that doing so allows the general issues to be brought out in a very clear way - far more clearly than in a "real world" domain, despite what one might think at first (from [17], p. 208)

What is meant by *fluid concepts* mentioned above? Here is an example suggested by Melanie Mitchell [26] that brings clarity to the concept of “fluid concepts”: Consider the question “Who is the first lady of Great Britain?” Here the concept under question is “first lady.” A straightforward interpretation of this “first lady,” the American way, is “wife of the president.” However, this definition would not work in case of Great Britain as it has no president. Rather than rigidly asserting that Great Britain has no first lady, most people treat the concept liberally, allowing some slippage. For instance, some people associate the prime minister of Great Britain to the President of America, and may say that Cherie (wife of Prime Minister Tony Blair) is the first-lady. The question of “first lady” is even more complicated when Mrs. Margaret Thatcher was the prime minister of Great Britain. Then many people were willing to say that the first-lady was Mrs. Thatcher's husband, Denis. Thus given certain pressures (mental) the concepts "president" and "wife" slip into "prime minister" and "husband" respectively as in Thatcher's case. This notion of fluidity - concepts slipping from one to another under specific mental pressure - is what is termed as "fluid concepts" mentioned above.

The Copycat architecture can be broken down into three components, the *Workspace*, the *Slipnet* and the *Coderack*. The current state of the problem is contained in the Workspace. The Slipnet is the knowledge-base for concepts. It is associative in nature. The Coderack is the staging area for the Codelets. Codelets are small programs that can perform simple tasks.

Figure 2.3 (taken from [30]) shows the overall architecture of Copycat including the instantiation of codelets from the codelet database.

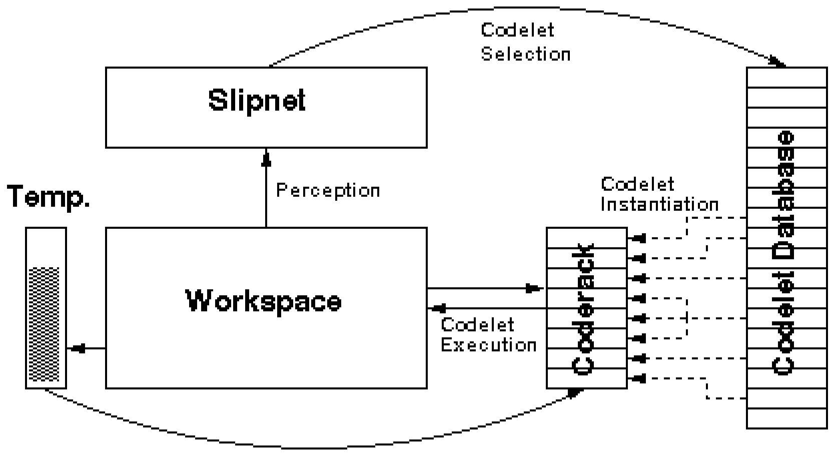


Figure 2.3: The Copycat architecture

The Workspace is the area where all temporary perceptual structures are built. It is the area where the initial problem is presented and where all the computational effort is focussed.

The Slipnet is an associative knowledge-base of domain specific concepts. As such it serves as Copycat's long-term memory. Nodes are joined by arcs which represent the affinity between them as their lengths. The main analogy making processing power of the Slipnet is the ability for suitably similar concepts to be used as substitutions for each other. This process was coined "conceptual slippage." Nodes can be attached to arcs which when activated can alter the arc lengths. In this way nodes can control the likelihood of conceptual slippages.

Associated with the Workspace is a measure of the current consistency of the built perceptual structures, called *temperature*. The temperature affects the types of codelets which the Coderack executes. The temperature controls the progression of Copycat from the initial data-driven explorations to the goal-driven processing which emerges later. The final temperature of the system is used as a measure of Copycat's satisfaction with a solution.

The Coderack is the issuing station for codelets. Codelets are the workhorses of Copycat, performing all the actions necessary to perceive a situation and perform actions within the Workspace. The Coderack issues codelets based on a weighted probability determined by the codelets "urgency." A codelet's urgency is assigned by whatever process brought it into the Coderack. Codelets can be added to the Coderack through two processes. Low-level, data-driven codelets are issued continuously by the system in an attempt to continually perceive and explore the current Workspace. Data-driven codelets are given a low urgency rating and effectively run as background processes. Goal-driven codelets are added to the Coderack by previously run codelets which assign an urgency rating according to their findings.

Why we cannot use the Copycat architecture in this research

As elegant as it may seem, we cannot use the Copycat architecture in our project because of the domain restriction. Unlike SME or other approaches that have a domain independent analogical reasoning component with only the knowledge being domain dependent, the Copycat architecture is closely tied to its microdomain. Hence, employing Copycat in this project would entail a lot of rework, which is not really the scope of this project.

Analogical Constraint Mapping Engine (ACME)

Psychologists Keith Holyoak and philosopher Paul Thagard have built a computer model of analogical mapping [20] called ACME (analogical constraint mapping engine). This model is similar to SME in that it uses representations of source and target situations given in formulas of predicate logic, and makes an analogical mapping consisting of pairs of constants and predicates from the representations. In fact, ACME has been tested on several of the same predicate-logic representations of situations that SME was given, including the water-flow and heat-flow representations.

This model takes as input a set of predicate logic sentences containing information about the source and target domains and it constructs a network of nodes (taking into account a number of constraints) where each node represents a syntactically allowable pairing between one source element and one target element[1]. A node is made for every such allowable pairing. For example, in the water-flow and heat-flow analogy mentioned above, one node might represent the *water* $\leftrightarrow$ *heat* mapping, whereas another node might represent the *water* $\leftrightarrow$ *coffee* mapping. Links between the nodes in the network represent "constraints"; a link is weighted positively if it represents mutual support of two pairings (e.g., there would be such a link between

[1]Here, "syntactically allowable" means adhering to "logical-compatibility," which specifies that a mapped pair has to consist of two elements of the same logical type. That is, constants are mapped onto constants and n-place predicates are mapped onto n-place predicates.

the *flow* $\leftrightarrow$ *flow* node and the *water* $\leftrightarrow$ *heat* node, since water and heat are counterparts in the argument lists of the two flow relations), and negatively if it represents mutual disconfirmation (e.g., there would be such a link between the *flow* $\leftrightarrow$ *flow* node and the *water* $\leftrightarrow$ *coffee* node).

The network can optionally be supplemented with a "semantic unit" - a node that has links to all nodes representing pairs of predicates. These links are weighted positively in proportion to the "prior assessment of semantic similarity" (i.e., assessed by the person constructing the representations) between the two predicates.

In addition, ACME has an optional "pragmatic unit" - a node that has positively weighted links to all the nodes involving objects or concepts (e.g., flow) deemed to be "important" ahead of time (again by the person constructing the representation).

Once all the nodes in the network have been constructed, a spreading activation relaxation algorithm is run on it, which eventually settles into a final state with a particular set of activated nodes representing the winning matches.

Sapper

The Sapper system was designed to address the problem of metaphor comprehension. Analogy-making plays a vital role in computational models of metaphor comprehension. Hence Sapper is essentially an analogical reasoner.

The primary difference between the Sapper's architecture and that of SME is that Sapper is a hybrid symbolic/connectionist model, which marries top-down structure recognition process with bottom-up spreading activation process [38]. On the other hand both SME and ACME (the connectionist flavor of SME) are top-down structure recognition processes.

Veale, the architect of Sapper, argues that SME and the like are algorithmically biased towards processing knowledge structures which have a nested structure or which are vertically codified. Figure 2.4 (taken from [38]) shows an example of vertical, nested or hierarchical structure suitable for SME. Figure 2.5 (also taken from [38]) shows an example of horizontal structuring, the type which is not suitable for SME. If these algorithms are given situations which have horizontal structuring, they are prone to intractability because they are factorially sensitive to the structuring [38]. Recognizing that most common metaphors have horizontal structuring, Veale developed Sapper to work in situations where SME and others would fail.

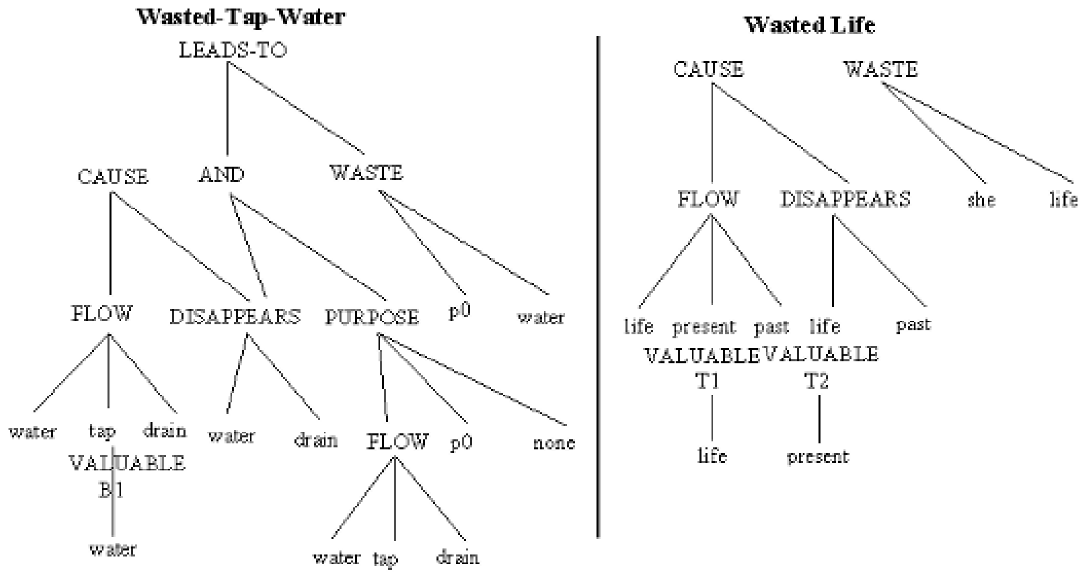


Figure 2.4: An example of vertically structured knowledge

This is one of the reasons why we decided to use Sapper rather than the more popular SME or ACME. In our application, the analogical reasoner is exposed to both kinds of structuring, horizontal as well as vertical. When CREAP has to compare two situations (like the dragon-gold scenario and hag-jane scenario) they are usually structured vertically. But when CREAP has to compare two objects (like sword and axe or dragon and troll), just like the noun-noun metaphors that Veale

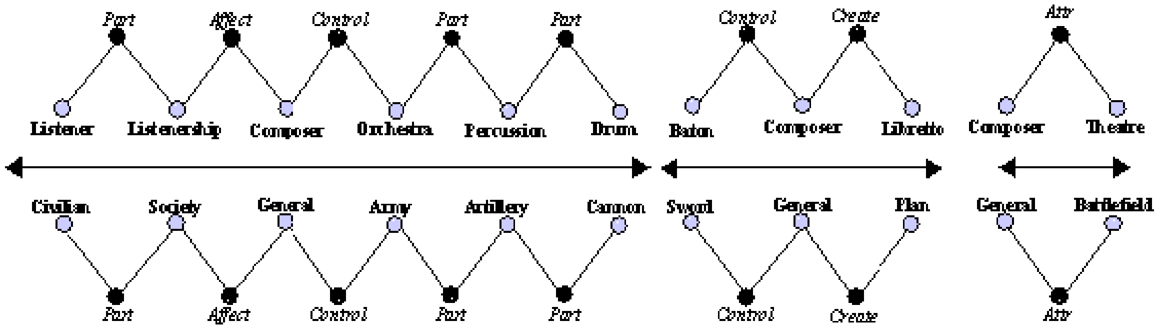


Figure 2.5: An example of horizontally structured knowledge

talks about, the natural way to represent these are in a horizontal fashion. Therefore if CREAP used SME then it would had failed in an important improvisation technique that we use in this project called "Precondition satisficing," which we will be seeing later on.

To sum up, Sapper addresses many issues that SME and others do not. Sapper can do whatever SME can do and also more. So Sapper subsumes the functionality of SME, plus gives us the power of handling noun-noun analogies. Therefore we have decided to use Sapper as our analogical reasoner.

Summary of computational models of Analogical reasoning

SME and ACME more or less fall in the same conceptual category. They differ little, fundamentally, and generally confirm to the following.

- Employs explicit and well-defined structure-mapping - if knowledge is structured (and we believe it is), then there must be some sort of structure-mapping that takes place to link up base-knowledge to target-knowledge.

- Based solidly on a robust psychological theory developed over almost two decades of empirical investigation.
- Generally applicable to all domains of analogical comparison.
- Requires hand-coding - can't produce it's own representations. Thus, depends on work done by humans.

Copycat, on the other hand, is fundamentally different from SME and ACME. Copycat, in a novel way, captures the notion of representation construction. SME and ACME do not address the representation construction issue. They assume that the situations presented to them are represented in a certain symbolic representation. Copycat is more autonomous. Once set up by humans in a specific domain (the letter-domain), it operates independently of humans. Domain specificity is a problem related to Copycat - it can only produce analogies in the limited letter-domain.

Sapper is a hybrid symbolic/connectionist model, which marries the top-down structure recognition process with the bottom-up spreading activation process. Sapper addresses many issues that SME and others do not. Sapper subsumes the functionality of SME, plus gives us the power of handling noun-noun analogies. Therefore we have decided to use Sapper as our analogical reasoner.

2.3 Simulation systems for intelligent agent designs

This Chapter on background would not be complete without discussing the simulation systems, such as V-World. V-World [4] is the simulation testbed system that was used to implement CREAP and its world.

In order to examine the behavior of an agent architecture, that architecture must be given a physical embodiment: the agent must be allowed to interact with an encompassing environment and display its behavior in light of stimuli from that environment. Such an embodiment is required during the development of an architecture for the purposes of feedback as well as for the purposes of testing the final design to illustrate both the competency and the limits of an architecture. For the purposes of this research, it was decided very early that a simulation system would provide the most appropriate environment for development for reasons that will be made clear in this section.

A simulation system for intelligent designs requires special elements not associated with general-purpose simulation languages and equivalent tools. For example, in order to be used for such purposes, a simulator must be generic: that is, it must provide the flexibility to accommodate an architecture that is undergoing constant modification, as well as the ability to represent a wide range of environments. *V-World*, specifically developed with the aim of experimenting with softbots, provides this kind of flexibility.

2.3.1 Simulated environments and intelligent agent research

Intelligent agents designed to perform in the real world should by definition be tested and evaluated in the real world. However, intelligent systems like CREAP continue to be developed using simulated testbed systems like V-World despite this obvious fact: typically, an agent is designed to demonstrate some aspect of intelligence and is tested in simulation, of course with the assumption that the demonstrated behavior will scale appropriately to the real world. However, the continuing disparity between the relatively small number of real world deployed systems and the large number of systems that perform only in small simulated *Blocks World* environments has come to be one of the strongest and most often cited arguments against the intelligent agent research conducted in simulated environments.

The difficulty with this suspicion is that it confuses problems that may arise from using a *simplified* environment with those that arise from using a *simulated* environment[1]. Simplified environments can never be like the real world, and only intensive analysis of the assumptions made in constructing these worlds can gauge the applicability of the results obtained from them. A simulated environment need not be an overly simplified model of reality. Indeed, simplification is not in and of itself a problem. Pollack (cited by [2]) goes to great pains to illustrate that not only is simplification not a problem, it is as absolutely necessary for experimentation in AI as it is in any science.

Despite this overall suspicion, there are many logical reasons for using V-World for testing intelligent agents such as CREAP. The foremost of these concerns the nature of the field of artificial intelligence itself. AI as science does not yet possess a wealth of broadly accepted theories that form a concrete foundation for ongoing research. Because of this, any broad research such as this will be hampered by being forced to deal with unsolved problems in peripheral areas. Almost any research in this area involves the integration of a number of areas like vision, knowledge representation, etc. with the necessity of solving open problems in those areas or making assumptions about the nature of those problems. Thus, if one develops a theoretical architecture for an intelligent agent like ours, unless one intends to solve *every* open problem in areas such as computer vision and robotics, one is forced to make (possibly incorrect) assumptions.

The use of a simulator in intelligent agent research is thus largely an issue of practicality: the large collection of interrelated problems encountered when implementing an intelligent agent necessitates the use of a simulator to aid in accounting for those pieces of the theory that are not yet complete.

In addition to basic limitations of research resources, there are many sound reasons for choosing to use simulators like V-World to examine the performance of intelligent agents like:

- Maintaining control and isolation.
- Maintaining consistency between trials.
- Ease of experimentation.
- Avoiding component unreliability faced by hardware technology.
- Maintaining a consistent agent interface.
- Providing an understanding of domain complexity.

2.3.2 V-World

V-World is a simulation system designed explicitly for the testing and examination of intelligent agent designs. Rather than simply providing a flexible domain, as is done in Ars Magna [8] or Tileworld [27], V-World provides flexibility in the simulation process itself. That is, rather than providing extensive domain parameters, V-World provides the parameters and functions necessary for the designers to define their own domain and interfaces for agents. This requires greater initial effort on the part of the designer, in order to define or modify a domain, but makes V-World much more widely applicable than most of the simulation systems.

Having been designed with generality in mind, V-World provides the following features:

- The ability to design the agent and multiple other entities in the agent's world.
- A clearly defined relationship between the agent and its world, including a model of timing for temporally extended actions.
- A concise and simple sensory and effectory interface between agents and the environment. This interface is provided for convenience, and may be diverged from if desired.

- The ability to control many aspects of the simulation itself, such as aspects of topology and ontology.

The V-World framework allows the designer to design a “world file” that defines the simulated world and an “agent file” that defines the agent. So the first task, as a designer, is to determine the characteristics of the simulated world and design it. Then we can go about designing the agent that embodies the behavior that we are looking for. Once the world and the agent files are in place, we can “load” the world and “run” the agent in that world.

The world designed and developed using V-World will have certain characteristics which we have described in section 3.1.1, where we discuss CREAP’s world. Likewise the agent developed using V-World has certain characteristics (with respect to sensory and effectory models) which we describe in section 3.2, where we discuss the properties of CREAP.

V-World is written in LPA WIN-PROLOG and runs under Microsoft Windows 95, 98, 2000, or Me. It includes a graphic display that allows the user to watch the agent as it moves about and interacts with other actors and objects in the world. V-World has been used in various other projects too [4]. In the next chapter we will see the utility of V-World when we discuss CREAP and its environment in more detail.

Chapter 3

CREAP and its domain

For developing a model of plan improvisation, the task domain must be sufficiently rich for the system to benefit from the improvisation without overwhelming the planning abilities of the system. In this chapter we describe the testbed domain in detail and sketch CREAP itself.

We are investigating a model for improvisational behavior as applied to a planning task, the task of creating and carrying out plans by a planning agent CREAP in a testbed domain. The task has many parts: *plan synthesis*, in which a plan is made deliberately which should get CREAP from the starting position to achieving the goal; second, *actual execution* of the plan that was synthesized; and last, *improvisation* in the event of a failure of the original plan. All three parts of the task are important. Without some sort of planning ahead CREAP would have no guidance in finding its way to achieving its goal. Without executing the plan it has no way to verify that its plan was correct. Finally, it should be apparent from what we have discussed that without improvisation CREAP would not be flexible in all the ways that we have discussed so far.

To get a feel for CREAP's world and what we expect of CREAP, let us consider an example world and an example situation. The example world is shown in figure 3.1. The world is fable-like, containing a princess, a dragon and various other entities.

In this example let us assume that the dragon is guarding the gold, and in order for CREAP to collect the gold it has to kill the dragon first. Further, CREAP's knowledge of this world indicates that dragons can only be slain by the sword; based

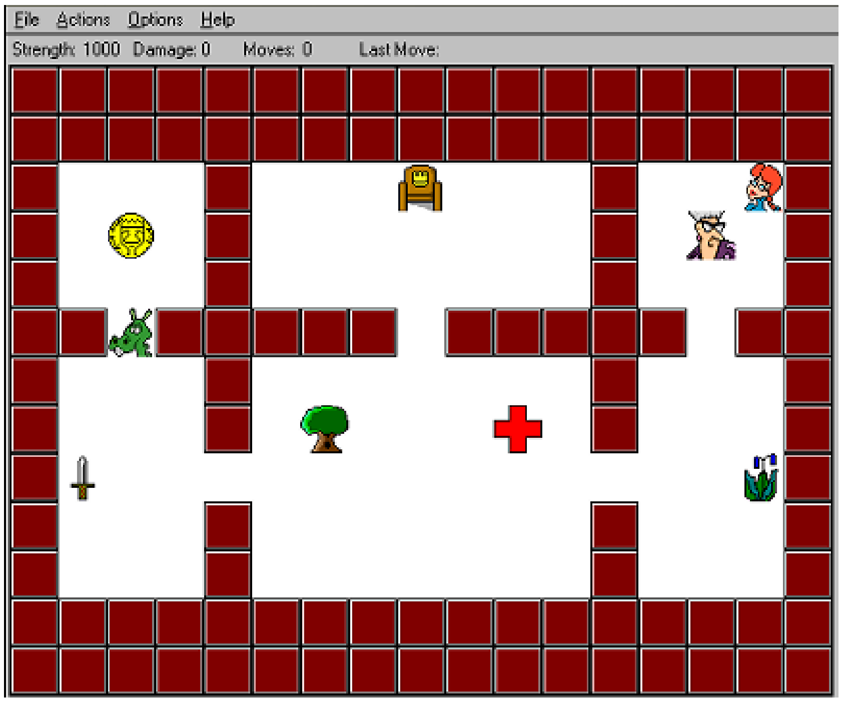

File Actions Options Help
Strength: 1000 Damage: 0 Moves: 0 Last Move:

Figure 3.1: A sample of CREAP's world

on this knowledge CREAP has to collect the sword before it goes to kill the dragon. Now, let us assume that CREAP is given "get gold" as the goal to be achieved. From its initial location, with "get gold" as the goal, CREAP uses its planner to synthesize a plan to achieve its goal, described by the high-level steps as:

1. go to sword
2. collect sword
3. go to dragon
4. kill dragon with sword
5. go to gold

6. get gold

CREAP then executes this plan in its world. If the plan succeeds without a hitch, CREAP should store this planning episode in its memory as successful, for future use. If CREAP encounters some problem in executing the plan it should improvise. For example, assume that CREAP's knowledge was slightly incorrect and when CREAP tries to kill the dragon with the sword, it fails. The world does not behave according to the expectations of CREAP. In such a situation we will fictionally explain two instances of CREAP's improvisation, one mundane and the other more creative.

In the first instance, when the sword fails to kill the dragon CREAP seeks an object similar to a sword. CREAP looks around and when it sees an axe lying there, it picks it up and hurls it at the dragon's throat. The ability to perceive the similarity between the sword and the axe led CREAP to improvise from "kill dragon with sword" to "kill dragon with axe." We call this type of improvising as "precondition satisficing," which will be explained further in Chapter 5.

In the second instance, when the axe fails too, CREAP seeks other ways to kill the dragon which is a stronger opponent. CREAP then sees a centipede and is reminded of a folktale about how an elephant was defeated by an ant:

```
...the ant entered into the elephant's giant nose and started to eat
the elephant from within. The elephant sneezed and sneezed...
Thus the giant elephant died and the ant emerged from the nose into
the splendor and glory of victory.
```

Drawing inspiration from this, CREAP picks up the crawling centipede and turns it into the dragon's worst enemy. Although only five inches long, the centipede destroys the dragon by crawling up to its nose, into its head, and killing it from within. This kind of creative improvisation requires grasping the *deeper analogy* between a folktale depicting a victory over the stronger enemy and the current situation, where CREAP is faced with a stronger enemy.

These two instances give an intuitive feel for how detailed or complex the domain should be. In setting up the actual domain implementation, we must determine a point on the spectrum of complexity which is both feasible and interesting for our task. A good implementation of the domain provides a level of complexity which is manageable, yet complex enough to lead to opportunities for exhibiting improvisation.

Beyond the richness of the representation of entities in the world, the domain should also include some dynamic elements. Elements that change and which are beyond the control of CREAP ensure that static knowledge is insufficient to capture the state of the domain, and provide more opportunities to exhibit improvisation.

In this chapter we will first describe the domain in which CREAP is situated, and we will discuss how our choices in creating this domain address the issues raised in the previous two paragraphs. We will then provide an overview of CREAP and CREAP's knowledge about its world.

3.1 The Domain

The domain chosen to test our theory is a testbed system called V-World. V-World allows us to design and develop custom-made worlds (like the world of CREAP we discussed above). It also allows us to design and develop our own softbots which are situated in these worlds. The nature of the world and the behavior of the softbot are entirely up to the designers, based on what theory they wish to implement and test.

In this chapter we will first describe CREAP's world in detail and then describe the characteristics of CREAP.

3.1.1 CREAP's world

CREAP's world is both spatially and temporally discrete. CREAP's world consists of many rooms corresponding to different geographical regions, separated by walls and connected by doors. Each room in turn is divided into distinct cells. Each cell can be empty, occupied by CREAP, occupied by any other entity in the world (like the sword, the princess or the dragon), or occupied by a wall. There are a total of seven rooms in CREAP's world: Castle, Field, Dragon's lair, Troll Bridge, Tower, Maze and Corridor (see figure 3.2). Each room is separated from the other by a wall and CREAP has to use the doors in order to get from one room to another.

Time is discrete too. Time is divided into "moves." In each move, CREAP and every other actor (animate object) have an opportunity to perform one action.

Apart from CREAP there are other entities that occupy this world. As the designers of the world, we get to choose what these other entities are and how they behave. These other entities are broadly classified along the following four axes.

- Animate or non-animate: Animate objects, also referred to as actors, are fairly simple reactive agents. Some of the animate objects move randomly in this world. Others can be made to follow CREAP wherever he goes or to guard a precious resource like gold. Such actors can be harmful, helpful or neutral. Dragons and hornets are examples of harmful animate objects. Non-animate objects are stationary. They are used by CREAP for various purposes. Some of the non-animate objects include sword, axe, tree, cross (the first aid kit), etc.

- Consumable or non-consumable: Consumable objects can be consumed by CREAP. When consumed, the object may have good or bad effects. For example, apples are consumable objects that increase the strength of CREAP. Hence they are beneficial.

- Collectable or non-collectable: Some objects are collectable and some are not. When CREAP collects a collectable object, it becomes a part of CREAP's inventory. Some of the collectable objects are sword, axe, and bird.
- Moveable or not moveable: Moveable objects are ones that can be pushed by the agent from one location to another. Apart from changing the location of the moved object, moving can have an effect on CREAP too. There is a reduction in CREAP's strength indicating the expenditure of energy.

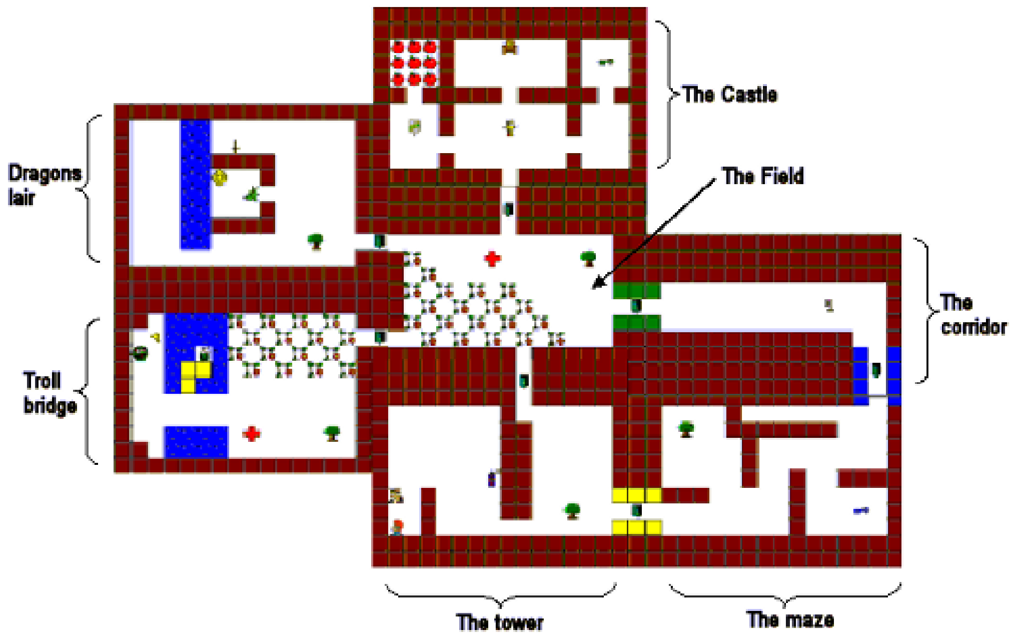


Figure 3.2: CREAP's world

3.1.2 Other entities

Apart from CREAP there are other entities that occupy CREAP's world. The following are some of the important ones.

- Jane (animate): Jane is the princess. She has been captured by the hag and is under a magical spell. In CREAP's world one has to bribe the hag with gold and the bird for her to release the princess and disappear.

- The hag (animate): a witch who has cast spells on the princess. She can often be bribed with gold and the bird.

- The dragon (animate, and harmful): The dragon is harmful to anyone who approaches it without a sword. However, if one approaches it with a sword, it is killed and it disappears. The dragon usually guards the gold.

- The troll (animate, and harmful): The troll is a mean character. It thumps anyone who approaches it, causing damage. The troll usually guards the bird and follows wherever the bird goes. The troll grows food on an island connected to the rest of the world by a bridge. This is the only source of food for the troll. If one uses an axe against the troll it is killed, and it disappears.

- The snail (animate): The snail moves randomly until it spots CREAP. From then onwards, it follows CREAP. It will freeze for a moment if pushed, giving CREAP a chance to escape. Snails are not harmful, per se, only a nuisance.

- Hornets (animate, and harmful): Hornets move at random. If one encounters a hornet it has to be avoided, else it stings and causes some damage. But hornets can be killed by using a bug-spray.

- The bird (animate): The bird moves at random. It can be collected. It does not cause any harm. It is sought by hag and guarded by troll.

- Fruits (non-animate, consumable, and beneficial): Fruits are consumable and if CREAP consumes a fruit it increases CREAP's strength.

- Fruit-trees (non-animate): Fruit-trees are the source of fruits. CREAP has to push on a tree for the fruit to fall on the ground, where it can then be consumed.

- Crosses (non-animate, beneficial): Crosses represent the first-aid stations. First-aid stations restore the health of anyone who pushes against them.

- Swords (collectable): A sword can be collected and used as needed. In this world, the sword is an effective weapon against the dragon.

- Axes (collectable): An axe can be collected. It is an effective weapon against the troll.

- Shields (collectable): A shield protects against thorns and against the troll.

- Thorns (non-animate, harmful): If one pushes or brushes against a thorn it can hurt, causing slight damage. This can be avoided if one has a shield.

These are the other entities with which CREAP has to interact. To summarize, this domain is a good one for testing the limits of CREAP's planning and improvisational behavior because of (1) the complexity of the information available to the system, (2) the dynamic elements included in the domain, and (3) domain extensibility. In this domain it is difficult to predict all the ways of responding to new situations, which is the requirement of the planning system (especially true if we keep CREAP's planning knowledge incomplete); hence the need for improvisation. New ways to respond, which are learnt as a result of improvisation, can be added to CREAP's existing knowledge. This is what we mean by *domain extensibility.* The extensibility of the domain makes working with the system tractable while allowing it to scale up. At the current time many possible complications and dynamic elements have been omitted, but can be included at a later time to test CREAP's robustness under increasing complexity.

3.2 Overview of CREAP

The focus of this project is to design and implement CREAP, a softbot which exhibits a certain behavior under the pressures of a certain environment that we have explained above.

Softbots are software agents that are situated in software simulated environments. They interact with their environment the same way an actual robot does in the real world. Softbots can be primitive randomly acting agents to sophisticated agents capable of reasoning and exhibiting intelligent behavior.

Just like actual robots have some sort of sensors to sense the environment around them, softbots also have sensors that tell them about the simulated environment. Again this could be as complicated as one designs the softbot.

Just as actual robots have effectors that translate symbolic actions to actual actions in the real world, softbots need ways to perform their actions in their environment. To handle this problem, most simulated environments take the responsibility of changing the state of the environment according to the symbolic actions issued by the softbots. In other words, softbots issue symbolic actions and the environment picks them up (by some mechanism) and changes the state of the environment according to the actions. We have similar mechanisms for perception and action built into CREAP which we shall discuss in this chapter.

CREAP is no mere reactive agent. It has to be a sophisticated knowledge-based agent in order to exhibit the kind of behavior that we desire. CREAP begins with some knowledge of its external world and of its own actions. It uses reasoning to maintain a description of the world as new percepts arrive, and to deduce a course of action that will achieve its goals. In this section we will also describe the extent of CREAP's knowledge.

But before that let's deal with two important parameters called *strength* and *damage*, which constitutes the life mechanism of CREAP.

3.2.1 Strength and Damage

CREAP, like all other softbots implemented using V-World, has two important parameters called *strength* and *damage* which constitutes its life mechanism.

CREAP's *strength* diminishes as it interacts with other entities and performs actions in its world. This models how humans expend energy in their daily lives and need to consume food to regain energy. CREAP is born with a default value of 1000 for strength. It gradually decreases as a function of CREAP's activity. When this value reaches 0, CREAP dies. There are certain objects in CREAP's world that can be consumed by CREAP to increase its strength value.

Another important parameter similar to strength is *damage*. This has value 0 when CREAP starts off. However, if it interacts with other objects with harmful properties the damage increases, which indicates that this interaction has caused some damage to the health of CREAP. This is one way to model situations like "attack by a dragon" - when CREAP goes near a dragon its damage increases. If this damage value reaches 100, CREAP dies. There are ways CREAP can decrease the damage already done, just like healing in people. Recall that CREAP's world has first-aid stations (the red crosses) and if CREAP pushes on this cross, its health is restored. (The damage value again becomes 0.)

3.2.2 CREAP's knowledge

CREAP belongs to the category of knowledge-based agents. In this category, agents can be seen as *knowing* their world, and *reasoning* about their actions. A knowledge-based agent needs to know many things in order to function: the state of the world, what it wants to achieve, and what its own actions do in various circumstances.

First of all, CREAP is required to navigate in its world a lot. How does it know where things are located in the world? To address this problem, CREAP has access to a detailed map of the entire world. With this it can navigate efficiently, using standard path-planning algorithms. There have been other projects where the map is not given to the agent and it is supposed to explore the world first, building the map on its own [4]. However, since the scope of this project is not exploration and search techniques, we simply hand-over the map to CREAP when it starts off. But CREAP's world is dynamic. Some entities change their position as time progresses. Therefore, the initial map handed to CREAP may not work later. To address this problem, CREAP is provided with a facility to update the map whenever it needs to use it. This process ensures that CREAP has information about the exact locations of other entities.

Ontological knowledge is as important as topological knowledge. It is not only sufficient to know where the dragon is located in the world, CREAP also needs to know what a dragon is in the first place. CREAP's domain model represents this ontological knowledge. In the previous section, where we discussed the domain, we presented the ontology of CREAP's world. The information that we presented was from the point of view of the designer of CREAP's world. This is the information we as designers know. But how much of that information does CREAP know? Surely, CREAP should be aware of its world in order to act. Just as we have a model of the world around us and we make our decisions based on this model, CREAP should have a model of its world too. We call this model the *domain model*. The domain model consists of facts about the domain like that the hag is guarding the princess, the sword is an effective weapon against the dragon, so on and so forth.

The *domain model* is declarative knowledge about CREAP's world. Building a domain model is laborious and time-consuming task. Therefore, we have engineered some facts and we let CREAP obtain some of the facts that are not included by querying the world (or nature). For instance, if it wants to know if a hornet is animate or not, it uses the query

```
?- animate(hornet)
```

which succeeds if hornets are indeed animate, else it fails.

CREAP's knowledge of what its own actions do in various circumstances is captured in the operator model. The *operator model* is a set of all the legitimate actions that CREAP can perform in its world, each subject to certain constraints. Where does this operator model come from? Similar to the domain model, a critical mass of operators are engineered a priori.

An important issue that we have to concern ourselves with in this research is "how detailed should this knowledge be?" In the earlier chapters we have made a strong case for these models to be incomplete in order to depict reality. We, as people, do not have complete and perfect models of the world around us. To assume that artificial intelligence systems should have complete and correct knowledge means restricting AI systems to idealized Blocks Worlds. Therefore in this project we have deliberately designed CREAP's domain and operator models to be incomplete and incorrect. This will bring out the need for improvisational behavior.

Since the domain and the operator models are used by CREAP's planner, we group them under the one heading *"planning-knowledge"* - the knowledge required by the planner. It is also important for this planning knowledge to be extensible over time. We are assuming that the planning knowledge is incomplete, which also means that we have to make provisions for it to be expanded as a result of CREAP's experience in its world. In the following chapters we hope to make it clear how CREAP's planning knowledge is expanded as a result of a successful improvisational act.

Finally, there is another important type of knowledge structure maintained in CREAP. It is stored in the *situation repository.* It is the same as *episodic memory* or *memory* as we have referred to it before. This knowledge is the basis for CREAP's improvisational behavior. In this memory CREAP maintains the previous planning and problem-solving episodes. We also call them "scenarios" or "situations." This type of knowledge structure is used by the analogical reasoning component of CREAP in order to make improvisation happen. So, when CREAP faces a plan failure, it may be reminded of one of the scenarios it has experienced before. It can use any relevant information from the past scenarios to address the present problem.

To summarize, CREAP has the following knowledge structures: domain model, operator model and situation repository, and the knowledge of the planning goal is hardwired into CREAP.

3.2.3 Perception and action

CREAP has limited and clearly defined abilities to perceive its world and act.

As we had mentioned earlier, CREAP's world is discrete and composed of many rooms, which in turn are made up of a grid of cells. Each cell can be occupied by CREAP or any of the other entities of the world, or it can be empty. From the cell in which CREAP is stationed, it can perceive a 5×5 square of cells around it. Nothing can block CREAP's view of this area. Figure 3.3 shows what CREAP can see from its current position.

The right-hand portion of figure 3.3 shows the actual part of the world and the left-hand side shows what is visible to CREAP.

Perception for CREAP is a little more than just looking. Perception is general awareness about itself and the world around it. Therefore perception involves:

- **Looking**: When CREAP looks around, using the `look/5` predicate, it perceives its own strength and damage, the set of objects it is carrying (if any) the direction of the last position the agent occupied, and the objects occupying the 25 spaces around it. The `look/5` predicate is as follows:

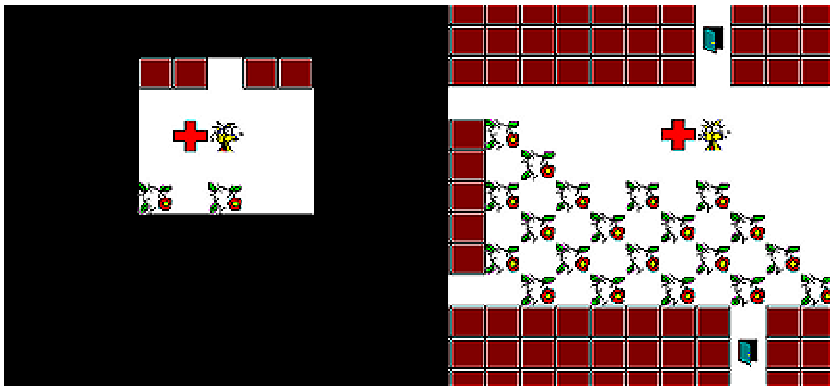


Figure 3.3: CREAP's field of view

```
look(Strength, Damage, Inventory, Last,[NWNW, NNW, NN,    NNE, NENE],
                                       [WNW,  NW,  N,     NE,  ENE ],
                                       [WW,   W,   CREAP, E,   EE  ],
                                       [WSW,  SW,  S,     SE,  ESE ],
                                       [SWSW, WSW, SS,    ESE, SESE]).
```

Each variable in the 5×5 grid in this pattern will become instantiated in a Prolog structure representing the type of object occupying that space. For example, if there is a cross to the left of CREAP as shown in figure 3.3, the variable W will become instantiated with the value cross in the structure above. Of course, the agent will always be in the position CREAP in the grid, and CREAP will always be instantiated to the atom 'a'. `Strength` and `Damage` will be instantiated to integers, and `Inventory` will be instantiated to a list of atoms representing whatever objects the agent is carrying.

- **Perceiving properties**: Some properties of the objects can be learned by querying the world (or nature). The four basic properties that objects can have are animate, collectable, consumable and moveable. If CREAP wants to find out the properties of an entity, say the dragon, then it queries nature using the four queries - `animate(dragon)`, `collectable(dragon)`, `consumable(dragon)` and `moveable(dragon)`. These queries succeed if dragon satisfies the respective properties.

The only primitive actions an agent can perform is to move one cell in any direction, including diagonally, to do nothing (sit) for one turn, or to drop an object in inventory (listed in the box at the left of the screen). However, an object or actor may be consumed, picked up, moved, attacked, or otherwise affected when the agent attempts to move into the space it occupies. How objects and actors respond to an agent depends upon the ontology of the particular world that has been loaded.

3.3 Summary

In this chapter we described the domain in which our theory is tested. We identified that (1) the complexity of the information, (2) the dynamic elements included in the domain, and (3) domain extensibility as the three important features that a domain should possess in order to test our theory. We also mentioned that V-World, a testbed system, allows us to create simulated worlds according to our requirements and also allows us to develop agents that can be situation in these worlds. We also discussed the nature of the world specifically created for CREAP. We noted the other important entities in this world. Then we gave an overview of CREAP and its interface with its world. We discussed CREAP's perception and action - means by which information from the outside world gets into CREAP and how information in CREAP is translated to action. In the next chapter we will discuss the architecture of CREAP.

Chapter 4

Architecture of CREAP

In this chapter we take a reductionist view to examine all the different components that constitute CREAP's architecture. We will be emphasizing the functionality of these components.

CREAP is a planning agent by design. In the previous chapter we discussed CREAP's domain of operation and the reasons for our choices in the way we designed it. In addition we gave an overview of CREAP and a brief description of CREAP's knowledge requirements and also its perception and action models. In this chapter, we will look at the components of CREAP in detail, examining how each works and what issues were addressed in implementing each.

CREAP's overall architecture consists of two main modules: the *planning module* and the *improvisation module*.

The *planning module* is responsible for creating high-level plans and executing them. In the best case scenario the planner is able to come up with a high-level plan, the execution is successful and the goal is achieved. However, things can go wrong. It is the responsibility of the planning module to also monitor its planning and execution stages. A failure in either of the two stages (planning failure or execution failure) causes the planning module to signal an interrupt. The planning and execution failures are further discussed in section 4.3.

When the planning module signals an interrupt, the *improvisation module* takes over. Depending on the stage of failure (planning failure or execution failure) the improvisation module tries to fix the situation accordingly. If the improvisation process is successful, the planning module can resume from where it had signaled an interrupt earlier. Hopefully the goal can be achieved this time.

The planning and improvisation modules consist of many components. Among these components a few are container components, emphasizing that they are some sort of repository for storing knowledge. Figure 4.1 shows the two modules and their components.

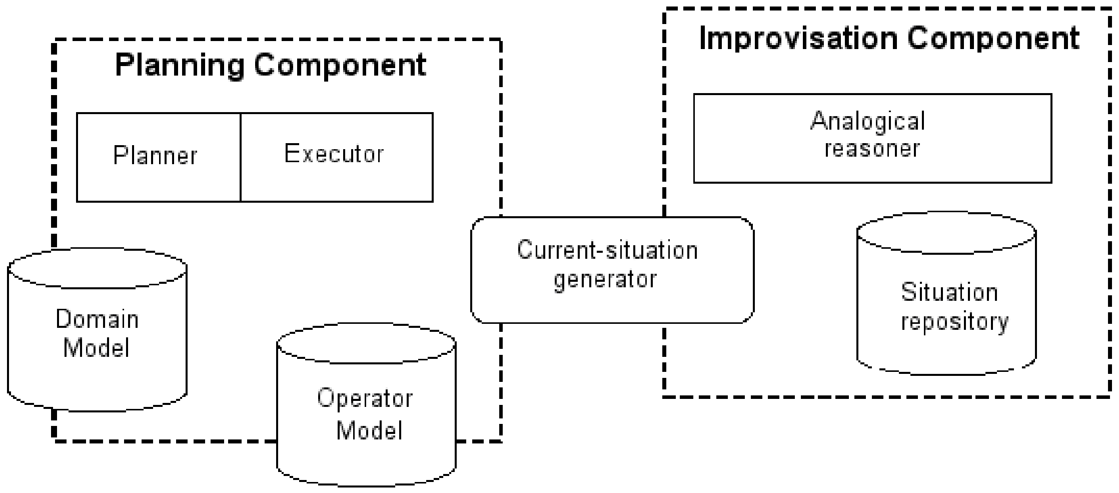


Figure 4.1: Planning and Improvisation modules with their components

The *planner* and *executor* are the heart of the planning module. Besides the planner and executor, the planning module also consists of two other container components: the *domain model* (stores factual knowledge about the world) and the *operator model* (stores the complex actions that CREAP can perform).

The SAPPER-based *analogical reasoner* is the most important component of the improvisation module. The analogical reasoner requires episodic memory, which contains a set of previously experienced situations. We call this the *situation repository.* The situation repository is the container component of the improvisation module.

The interface between these two modules is the *current-situation-generator* component. When the planning module fails, this component takes stock of the situation and creates a sort of a report of what went wrong and under what prevailing conditions of the world. This information is passed to the improvisation module.

In this section we will examine all these components and look at the function of each in detail. This is a reductionist view of CREAP's architecture. Because of the focus on the functioning of the individual components, the high-level picture that emerges of the overall process might be a little hazy. However, in the next chapter we will discuss how all these various components come together and interact. We hope that by the end of the next chapter one should be able to get the big picture.

4.1 The Planner

The function of CREAP's planner is to generate a high-level plan given the goal. The problem of designing systems that can plan has been of considerable interest in the field of artificial intelligence, especially automated reasoning. A planning system is one that can describe a set of actions that will later allow it to achieve a desired goal.

The way a classical planner works is entirely different from how a memory-based or a reactive planner would work. Since we are using a classical planner, we will restrict ourselves to the realm of classical planners. Henceforth, whenever we use the term "planner" it refers to a classical planner. Therefore when we say that the planners use an operator schemata or some such thing, it means that the classical planners use it and not "all" types of planners.

4.1.1 The planning terminology

In order to understand some of the concepts related to planning research, we need to familiarize ourselves with terms that are commonly used in planning literature.

An AI planning system is responsible for synthesizing a *plan* which is a possible *solution* to a specified planning problem. A *planning problem* is characterized by an initial and a final state description. The *initial state description* describes the world as it is before the planning execution begins. The *final state description* (goal state) describes the world as it is supposed to be after the plan is executed.

Actions, known as *operators*, are what are at the disposal of the planner to go from the initial state to the final state. The job of the planner is to choose the right set of operators, in the right order, for a given problem specification.

Operator schemata describe actions or operators in terms of their *preconditions* and *effects*. Each operator schemata stands for a *class* of possible actual actions which are created by instantiating the corresponding schemata by providing bindings for schemata's variable(s). These variables in operator schemata are replaced by constants to derive *operator instances* that describe specific, individual actions. The term *operator* is generally used to denote both operator schemata and operator instances.

A typical operator would consist of the following three components:

- **Action description**: This represents the name of the possible action that a planner can take along with its parameters.
- **Preconditions**: A conjunction of facts that must be true for the operator to be applicable, i.e., a partial description of a state of the world.
- **Effects**: A conjunction of facts that are expected to hold after the application of the operator.

For example, consider an action take-block which when executed will take a block from `loc(x1,y1)` to `loc(x2,y2)`.

```
Action/Operator: take-block
Preconditions: at(loc(x1,y1))
Effect: at(loc(x2,y2))
```

In our particular implementation of the planner this operator schema would be represented as:

```
preconditions(take-block, [at(loc(x1,y1)]).
effect(take-block, at(loc(x2,y2)).
```

A planner that searches forward from the initial state to the final state looking for a sequence of operators to solve the problem is called a *progression planner*. On the other hand, a planner that searches form the goal state to the initial state is called a *regression planner*. Regression planners are considered more efficient because, typically, there may exist many actions that may be executed in the initial state (i.e., many actions might have the initial state as the precondition), but there are usually few actions that have the effects matching the goal state.

A *state-planner* is a planner which searches through a space of possible states (e.g., STRIPS). A planner may alternatively search through a space of plans, in which "states" are partial solutions, i.e., partially complete plans. Such a planning technique is known as partial-order-planning and the planners of this type are referred to as *partial-order-planners*, POP for short.

CREAP's planner uses a POP. Hence we will be discussing POP in more detail.

4.1.2 Partial-order planner (POP)

CREAP's planner is a domain-independent planner. This means that if CREAP is moved to another world, the planner does not change. What do change are the planning knowledge and the executor part.

A partial-order planner (POP) is one that searches through a space of plans rather than a space of possible states. In POP "states" are partial solutions, partially complete plans. The idea is to start with a simple, incomplete plan called a "partial plan." This plan is extended until a complete plan is obtained. The operators in this kind of planning are not actions as mentioned previously, but they are operators on plans - such as adding a step, imposing an ordering on the restriction on the steps or instantiating a previously unbound variable. The goal state is the final plan. Thus, POP is a planner that searches through a space of plans. We use POP to provide CREAP with plan generation capabilities.

Further, some partial-order planners use the principle of least-commitment, in which a planner makes a commitment to the details of the plan only as necessary. The details of the plan here refer to such things as ordering information and binding of variables to values. A planner that represents partial plans, in which some steps are ordered with respect to each other and some steps are unordered, is referred to as a *partial-order* or *nonlinear* planner. Examples of nonlinear planners are NOAH and UCPOP [33]. We can derive a totally-ordered plan from a partially-ordered plan by a process called *linearization*, which imposes an order on the partially-ordered-plan.

CREAP's planner is based on the POP algorithm mentioned in Russell and Norvig [33].

Formally, a plan may be defined as a data structure consisting of the following four components:

- A set of plan steps. Each step is an operator for the problem domain.
- A set of ordering constraints, each of the form $S_i \prec S_j$, implying that step S_i occurs before S_j , but not necessarily immediately before.
- A set of causal links or protected links. A causal link is a three-tuple. It may be represented as (S_i, S_j, c), where step S_i achieves condition 'c' for step S_j and, therefore, c must be protected from the time S_i is executed until the time S_j is executed.

- A set of variable binding constraints. Each variable is of the form $var = x$, where *var* is a variable name and x is a constant or another variable.

A plan can be complete or not. A *complete plan* is one in which every precondition of every step is achieved by some other step. Likewise, a plan can be consistent or not. A *consistent plan* is a plan in which there are no contradictions in the ordering or binding constraints. We call a plan that is both complete and consistent a *solution* to the planning problem.

So, the purpose of CREAP's planner is to look at the goal given to CREAP as a planning problem and come up with a solution.

As we can see, the planning algorithm operates on the operators. For the operators to be applicable their preconditions need to be satisfied. These preconditions can be satisfied either by the application of the other operators, or they could be facts which are true in the planning environment or the domain. Therefore the set of operators and the facts that are true in the domain constitute the knowledge of the planner. We shall be discussing the knowledge required for CREAP's planner in the following sections.

4.2 The Executor

The plan synthesized by CREAP's POP consists of a set of symbolic descriptions of actions to perform in order to achieve the goal. Here is an example plan:

1. go to sword
2. collect sword
3. go to dragon
4. kill dragon with sword

5. go to gold

6. get gold

These actions will have to be executed in the world. Plan synthesis is like making a list of things to do and execution is like carrying out those actions in the list.

Recall that, the only primitive action an agent can perform is to move one cell in any direction, including diagonally. This means that a high-level action such as “go to sword” has to be translated into a set of primitive actions or moves.

The plan executor is responsible for taking a high-level plan and generating a set of low-level moves and executing them. For example, a high-level action such as “go to sword” might get translated to [sw,w,n], which says that the agent has to travel one cell to the south-west, one cell to the west and then one cell to the north to reach the sword.

Making low-level moves from high-level action descriptions is NOT domain-independent. It involves domain semantics and a precise map of the world in which CREAP is situated. Therefore, CREAP’s planning module consists of a domain-independent POP and a domain-dependent executor.

4.3 A note on planning and execution failures

In CREAP we have seen that the POP generates a high-level plan. The plan executor then takes this high-level plan and executes it. However, there could arise circumstances in which POP fails to come up with a high-level plan. We refer to this as *planning failure.* On the other hand, POP may have a high-level plan but the executor could fail during execution of this plan. We refer to this as *execution failure.*

Let us first focus on the planning failure, which we define as follows: where the planning knowledge is correct and the planning algorithm is complete, a planning failure occurs if there exists a plausible plan P for achieving a certain goal

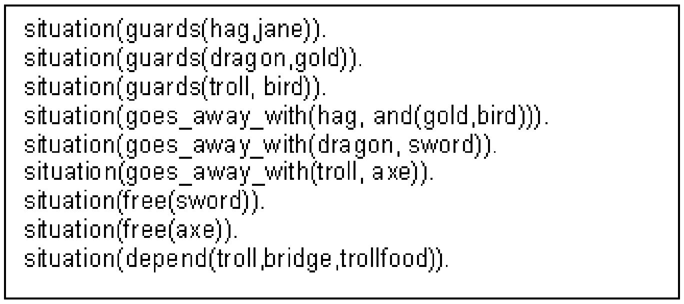
G in a given situation S, but the planner fails to find P. Given that this definition assumes correct planning knowledge, the only other factor that could lead to planning failure is incomplete planning knowledge. In CREAP, when the planning knowledge is incomplete the planner cannot find any action in the operator model to satisfy a precondition of another action in the partial-solution. This leads to a planning failure.

While incomplete planning knowledge leads to a planning failure, at least according to the way we have defined it, incorrect planning knowledge leads to an execution failure. For instance, an action might not have the intended effect during execution, or a fact in the domain model which was assumed to be true might turn out to be false. These cases are two instances of incorrect planning knowledge leading to execution failure. And even if a plausible high-level plan is generated by the planner and the planning knowledge is correct, execution may still fail due to the unexpected changes in the external world. That is why real world planning is so hard.

4.4 The Domain model

For CREAP's planner to succeed, CREAP must have a pretty good idea about all the other characters that are involved, what their needs are, whether they are harmful or beneficial, how to react when it encounters any one of them, etc. In other words, it must have a *domain model* of its world. The domain model is a container component which stores facts about the world.

How does CREAP acquire this knowledge of its world? We knowledge engineer some of the facts about its world into CREAP. Figure 4.2 shows some of the engineered facts, encoded as situation clauses as part of the domain model. But this does not constitute the entirety of CREAP's knowledge about its world. CREAP also acquires some of the facts, such as properties of objects, as a result of its perception (see previous chapter).



```
situation(guards(hag,jane)).
situation(guards(dragon,gold)).
situation(guards(troll, bird)).
situation(goes_away_with(hag, and(gold,bird))).
situation(goes_away_with(dragon, sword)).
situation(goes_away_with(troll, axe)).
situation(free(sword)).
situation(free(axe)).
situation(depend(troll,bridge,trollfood)).
```

Figure 4.2: A few engineered facts in the domain model of CREAP

The assumptions we make about the domain model is quite significant for this project. Most systems that use classical planners make the assumption that this domain model is complete and correct. A complete and correct domain model is one where "all" that is true in the world is encoded in the domain model, and "all" that is encoded in the domain model is indeed true in the world. We have already discussed this issue and we base CREAP's domain model on the assumption that it can be incomplete and incorrect.

The significance of the domain model needs some more emphasis. If the domain model is partial (or incomplete), CREAP is not guaranteed to come-up with a successful plan. For instance, if the domain model does not include `guards(hag, jane)` - the fact that the hag is guarding the princess - then it cannot include the `bribe(hag)` action into the plan as `bribe(hag)` operator has `guards(hag, jane)` as the precondition. And, since there is no other way to free the princess, CREAP might fail to construct a plan to free the princess.

Similar is the case for an incorrect domain model. If, for instance, CREAP's domain model says that there is a free axe in some part of the world, CREAP might make a plan based on this knowledge, go to pick up the axe, but find that it is not there, or it is unusable. Such situations are common in the real world and sometimes jeopardize the execution of the planner.

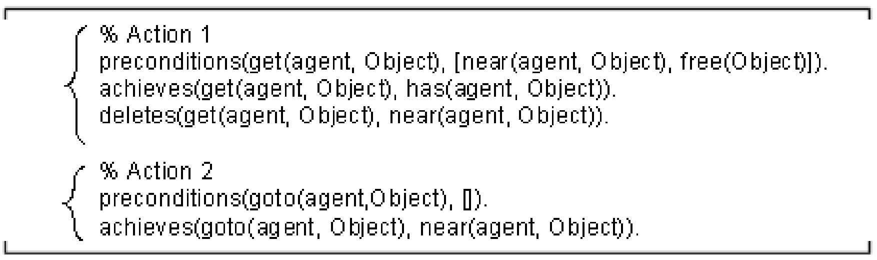
We intend to bring out many such situations in CREAP's span of activity. Every such instance of plan failure is an opportunity for CREAP to exhibit its improvisational behavior. Therefore, we do not bother about putting each and every true fact into CREAP's domain model. In fact, we have deliberately tried to introduce some errors into CREAP's domain model. Thus, CREAP's domain model was engineered to be incomplete and incorrect.

Even if the domain model were complete and correct when CREAP starts off, it's not guaranteed to achieve its goal because of the randomness involved in its world. Certain events in the world occur at random and CREAP cannot predict these based solely on its domain model. Using conditional/contingency planning techniques CREAP's planner can be enhanced to construct plans that account for some of the random events or contingencies that could arise. But there is a limit to how much one can plan in advance taking contingency into account. However, that enhancement is not dealt with in this project. In any case, CREAP is not guaranteed to achieve its goal.

4.5 The Operator model

The *operator model* consists of knowledge about various actions that CREAP can perform in its world, under what conditions and what its consequences are. In other words, the operator model is a set of all *operator schemata* (see section 4.1.1). This knowledge is encoded in a specific format which CREAP's planner can understand. The operator model is also a container component, like the domain model, but it consists of knowledge about CREAP's own actions rather than the external world.

Ideally speaking, this knowledge should also be a part of the domain model. How does the knowledge about some situation, say “hag is guarding princess Jane” differ from the knowledge about an action that CREAP can perform, say “bribe the hag to free princess Jane?” Thus speaking, there is no need for two separate domain and

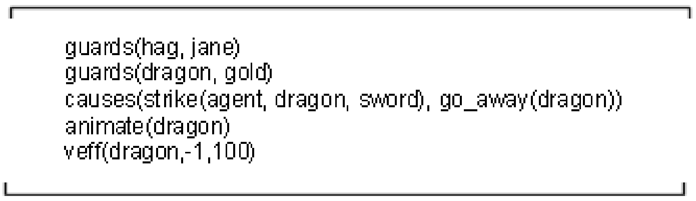


```
% Action 1
preconditions(get(agent, Object), [near(agent, Object), free(Object)]).
achieves(get(agent, Object), has(agent, Object)).
deletes(get(agent, Object), near(agent, Object)).

% Action 2
preconditions(goto(agent,Object), []).
achieves(goto(agent, Object), near(agent, Object)).
```

Figure 4.3: STRIPS-style representation of actions

operator models. However, the planner requires the operators to be specified in a particular ‘STRIPS-like’ representation, shown in figure 4.3. Compare this to the predicate representation of situations in figure 4.4, a part of the domain model.



```
guards(hag, jane)
guards(dragon, gold)
causes(strike(agent, dragon, sword), go_away(dragon))
animate(dragon)
veff(dragon,-1,100)
```

Figure 4.4: Predicate representation of situations in domain model

We could combine all these into one knowledge-base and call it the domain knowledge. However for the sake of convenience and clarity, we have put all the actions that CREAP can perform into a separate container, and called it the operator model. All the other facts go into another container called the domain model.

CREAP cannot start off with an operator model that is empty. The issue of obtaining a critical mass of these planning operations arises. There are two ways of obtaining the initial set of operators: *knowledge engineering* and *machine learning* techniques. Although the machine learning approach seems tempting, we have adopted the knowledge-engineering approach to keep it simple.

Approaches to automatically obtaining the initial set of planning operators can be seen in recent planning literature. Wang [34] (planning while learning operators) describes an approach wherein planning actions are learnt by observing expert solution traces, and further refining these operators through practice by solving problems in the environment in a learning-by-doing paradigm. Gil [14] also presents another approach to automated acquisition of domain knowledge for planning via experimentation with the environment. All these approaches can be implemented into CREAP's existing architecture given the flexibility of the architecture and the operator-based representation.

All the components of our system that we have discussed so far (the domain model, the planner, the operator model) are usually part of any traditional planning system. The components discussed henceforth are unique to CREAP's architecture. They are required because we are nullifying the unrealistic assumption of a complete set of planning operators/actions often made by traditional planning systems. This is justified given that planning actions/operators are knowledge-engineered, and the problems of formulating the domain knowledge for planning are well known. It is safe to assume that in certain planning situations the normal planning algorithm breaks down because it cannot find an operator/action that achieves a desired effect. The questions *"what is the best that we can do when a planning system fails to come up with a plan because it does not know of an action that will produce a desired effect"* and *"can we look at the past (earlier situations) and guess a new action that may achieve the desired effect in the current situation?"* have motivated this work and the inclusion of the the subsequent components into CREAP's architecture.

4.6 The Current-situation-generator

The current-situation generator is a component responsible for what happens immediately after the planning module fails. Let us take a closer look at the situations when the planning module fails. The planning module can fail at either the *plan synthesis* stage or at the *plan execution* stage.

Failure during plan synthesis

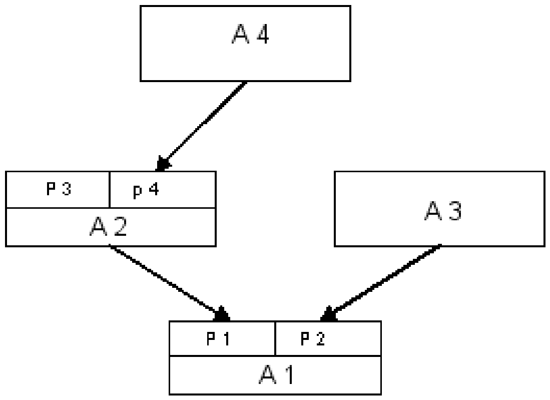


Figure 4.5: Incompleteness in operator model

As we have already mentioned in section 4.3, failure at the plan synthesis stage is because of the incomplete planning knowledge. For instance, consider the partial-solution which has been generated by the planner in the course of a planning activity shown in figure 4.5. Notice that there are four actions: $A1$ with preconditions $P1$ and $P2$, $A2$ with preconditions $P3$ and $P4$, $A3$ and $A4$. $A2$ and $A3$ satisfy $P1$ and $P2$. $P4$ is satisfied by $A4$, and $A3$ and $A4$ do not have any preconditions. This leaves $P3$ dangling. For this partial-solution to become a complete plan: (1) the planner has to either find another operator that satisfies P3, or (2) backtrack from this partial solution and try to find another operator that satisfies $P1$, thus discarding $A2$ altogether. If none of these can happen, the planner fails. Therefore, if the planner fails at the plan synthesis stage it is because it could not satisfy a precondition of some operator in the partial-solution and there were no alternatives to this operator. This is true because of the way a POP works, or any regression planner that starts from the final goal and computes the plan backwards.

Failure during plan execution

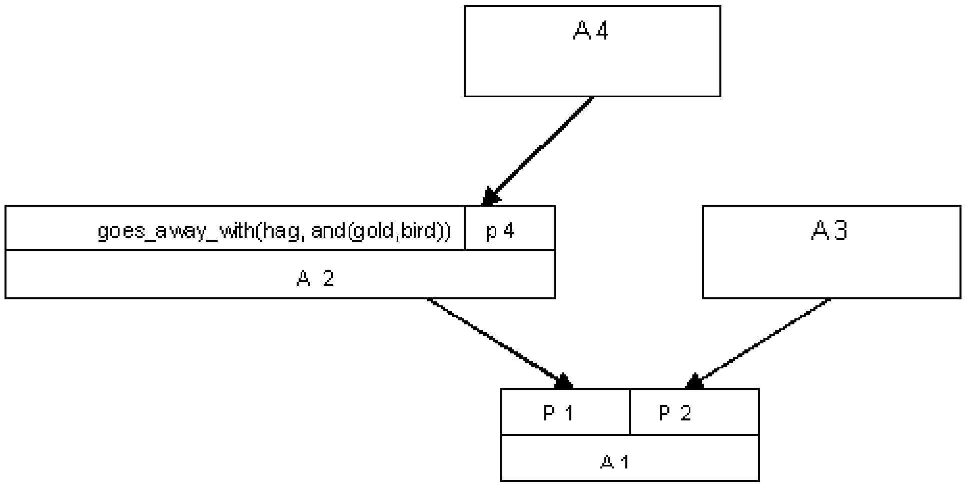


Figure 4.6: Incompleteness in operator model

The planning module can also fail at the plan execution stage. As we have mentioned earlier, a failure at the execution stage indicates the incorrectness in the planning knowledge. Consider an example solution in figure 4.6.

This solution can fail at the execution stage if either

(1) the `goes_away_with(hag, and(gold,bird))` precondition which was assumed to be true (based on the domain model) was not actually true in the world, or (2) operators $A2$, $A3$ and $A4$ did not have their intended effects, failing to satisfy preconditions $P1$, $P2$ or $P4$ respectively. The later is the uncertainty problem, where a particular action sometimes produces an intended effect and sometimes not. That is, a particular action produces an effect with a probability less than 1.

We are not dealing with the uncertainty issue here and we assume that all the actions have their intended effects at all times. Therefore if the execution fails it has to be because some fact that is true in the domain model of CREAP fails to be true in the actual world. This is the incorrectness problem.

Planning process interrupted

Any sort of a failure, either at the plan synthesis stage or at the execution stage, is considered as an interrupt in the planning process. The interrupt hands-over control to the current-situation-generator. The current-situation-generator makes a situation report. This report will include the precondition that failed, and some associated facts gathered from the domain model. As an example, assume that the planner has included the following action in the partial plan so far:

```
[ preconditions(take_to_throne(agent,jane),[near(agent,jane),free(jane)])
  achieves(take_to_throne(agent,jane),on_throne(jane)) ]
```

The above `take_to_throne` action has two preconditions `near(agent, jane)` and `free(jane)` to be satisfied. Further let us assume that the action `goto(agent, jane)`, which is already a part of the operator model, satisfies the precondition `near(agent, jane)`. Given this, and assuming that `free(jane)` is not a fact true in the domain model, if the planner fails at the planning stage it is because it could not find an action that satisfies the precondition `free(jane)`.

The function of the current-situation-generator is to make a report of the failed situation. Therefore in the above event of a plan failure, the current-situation-generator includes in its report `free(jane)` as the goal that the planner was trying to achieve when it failed. It also consults the domain model to find out more about jane. It finds `guards(hag, jane)` as a fact. It tries to finds out more about the hag from the domain model. It gets `goes_away_with(hag, gold)` as a fact. Based on these it generates the situation report

```
[  [goal, 0.5, free(jane)],
   [fact, 0.5, goes_away_with(hag,gold)],
   [fact, 0.5, guards(hag, jane)]
]
```

But what is the use of such a situation report? It serves as an input to the analogical reasoner. The analogical reasoner looks at this situation report and tries to find a new action that could satisfy the goal `free(jane)`, based on its background knowledge.

4.7 The Analogical reasoner

The analogical reasoner is the heart of the improvisation process. The function of the analogical reasoner is to look at a failed sitation and try to infer additional knowledge which could be helpful in fixing the failed plan based on the analogical reasoning and background knowledge. In this section we will see how.

The input to the analogical reasoner is the situation report of the failed situation that was generated by the current-situation-generator. We have already seen how Sapper, the system we have used as CREAP's analogical reasoner, works. To put it very simplistically, the analogical reasoner compares the current situation report (*the target*) with other situations stored in a separate repository and retrieves a situation (*the base*) that is structurally very close to the target. Based on this structural similarity, we can map entities in the target and the base on a one-to-one basis, and we can also infer new predicates from the base. One or more of these inferred predicates holds the potential to become new actions in the target situation.

Let us study this more closely with the help of the following example. Assume that the planner fails and the current-situation generator assembles the following report, which we have already seen before:

```
[  [goal, 0.5, free(jane)],
   [fact, 0.5, goes_away_with(hag,gold)],
        [fact, 0.5, guards(hag, jane)]
    ]
```

This report is the input to the analogical reasoner. The analogical reasoner compares this report with the knowledge stored in its memory. In addition let us assume that CREAP's episodic memory (situation repository) has the dragon-gold situation stored, shown in figure 4.7. This means to say that CREAP has directly or indirectly

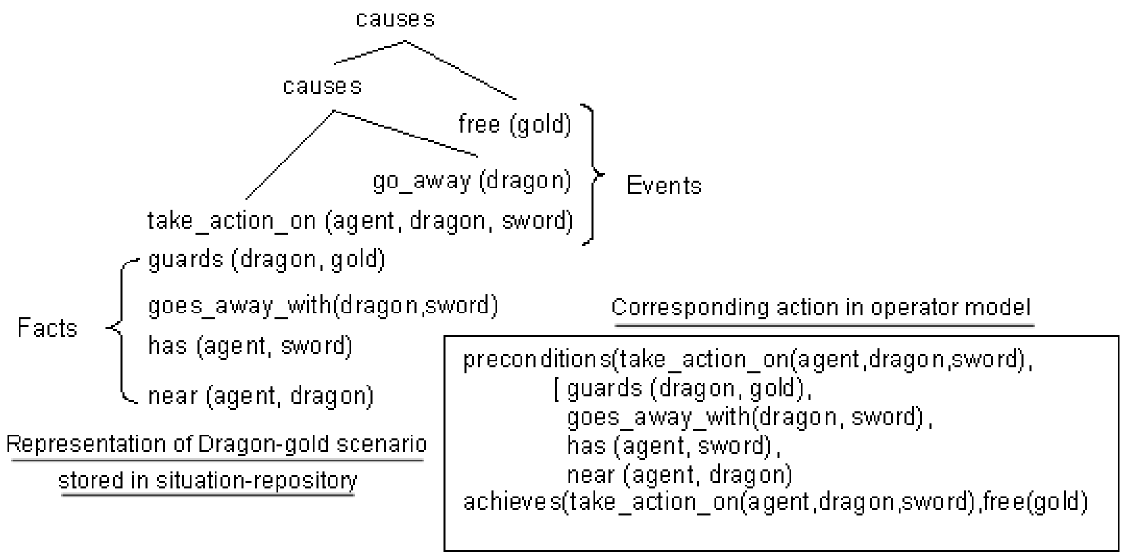

causes
causes
free (gold)
go_away (dragon)
take_action_on (agent, dragon, sword)
Events
guards (dragon, gold)
goes_away_with(dragon,sword)
has (agent, sword)
near (agent, dragon)
Facts
Representation of Dragon-gold scenario stored in situation-repository
Corresponding action in operator model
preconditions(take_action_on(agent,dragon,sword),
[guards (dragon, gold),
goes_away_with(dragon, sword),
has (agent, sword),
near (agent, dragon)
achieves(take_action_on(agent,dragon,sword),free(gold)

Figure 4.7: The Dragon-gold scenario

come across this situation before.

Based on the structural similarity between the current situation report and the dragon-gold scenario, the analogical reasoner maps the predicates as show in figure 4.8. Based on the mapping the entities shown in figure 4.9 are considered analogous.

From the structure-mapping and candidate inferences, the analogical engine constructs a new scenario, shown in figure 4.10.

From the above inferred scenario, we can easily formulate a new planning operator/action by conjoining the facts as the preconditions, and by making the final

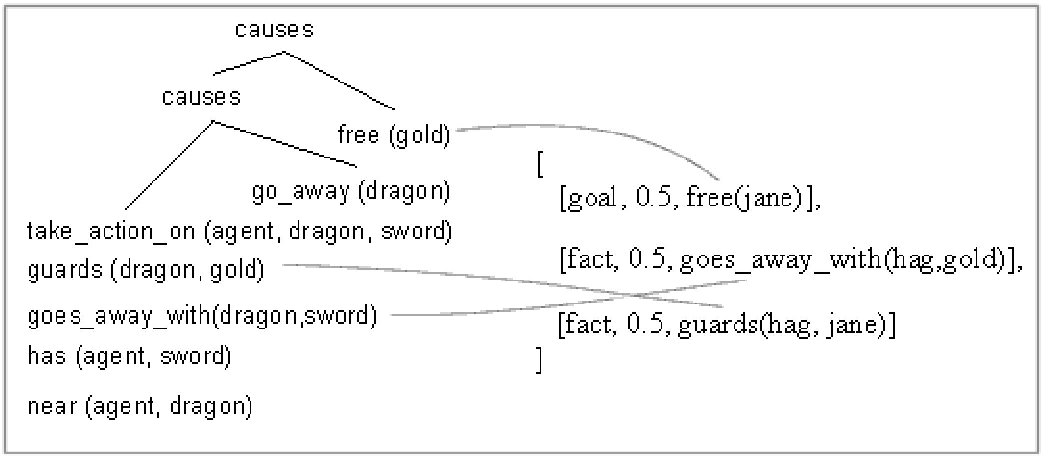

causes
causes
free (gold)
go_away (dragon)
take_action_on (agent, dragon, sword)
guards (dragon, gold)
goes_away_with(dragon,sword)
has (agent, sword)
near (agent, dragon)
[
[goal, 0.5, free(jane)],
[fact, 0.5, goes_away_with(hag,gold)],
[fact, 0.5, guards(hag, jane)]
]

Figure 4.8: Predicate mapping between the the two scenarios

situation report (target)	**dragon-gold scenario** (base)
agent	agent
gold	jane
dragon	hag
sword	gold

Figure 4.9: Analogous entities

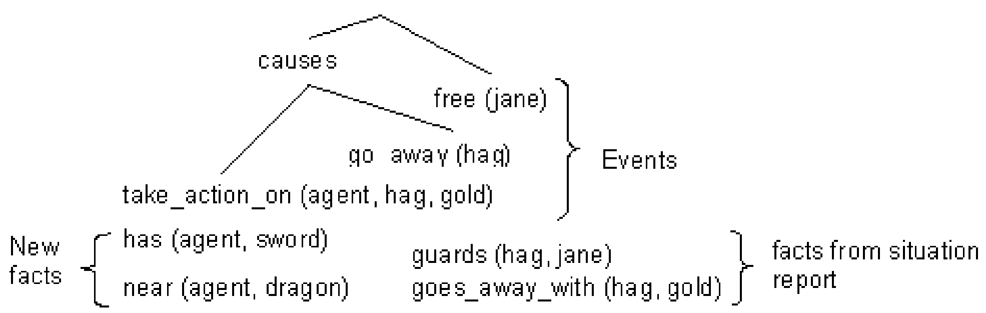
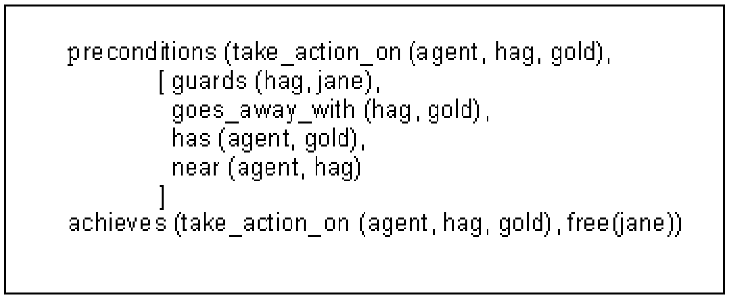


Figure 4.10: New scenario constructed based on analogy

cause (also the goal of the situation report) the effect. Therefore we have constructed a new planning operator by analogical reasoning as shown in figure 4.11.



```
preconditions (take_action_on (agent, hag, gold),
         [ guards (hag, jane),
           goes_away_with (hag, gold),
           has (agent, gold),
           near (agent, hag)
         ]
achieves (take_action_on (agent, hag, gold), free(jane))
```

Figure 4.11: Newly formulated operator

Let's summarize how we went from a failed situation to the formulation of a new operator:

- CREAP failed to plan initially because it did not know how to `free(jane)`
- But, CREAP had earlier encountered a similar, but not an exactly identical, dragon-gold scenario before. The dragon-gold scenario included `free(gold)` as the goal.
- In the dragon-gold scenario, CREAP had used a sword on the dragon to cause `go_away(dragon)`; since `guards(dragon, gold)`, CREAP had achieved `free(gold)`.
- Based on analogy, CREAP knows that the two situations are similar but the entities are different.
- CREAP maps the right entities in the two scenarios. It infers that by offering the hag the gold, it can cause `go_away(hag)`; and since `guards(hag, jane)` is a fact, it can cause `free(jane)`.

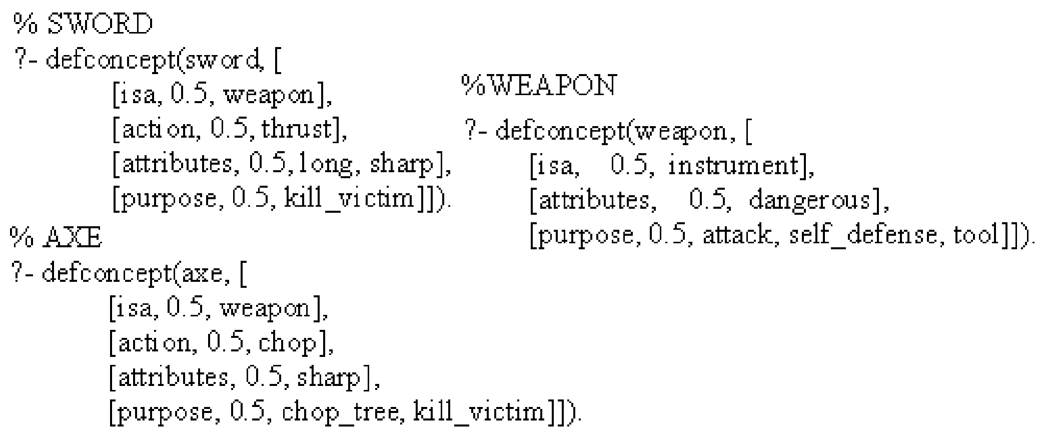
We can now add this newly formulated action into the planning knowledge-base. With the inclusion of this new action into the operator model and assuming that all the other actions satisfying all the preconditions are present in the planning knowledge-base, the planning will now succeed. But we cannot yet confirm the correctness of the new action. Since this was our best guess, based on an earlier experience, it might fail during execution. We can only confirm its correctness at the execution stage, when it achieves the desired effect in the operating environment.

4.8 The Situation repository

The situation repository is a collection of background knowledge. This knowledge is separate from the planning knowledge which consists of the domain model and the operator model. The situation repository contains various scenarios encountered in the past (episodic memory) and also some background information about objects in the domain. It is stored in a representation that is consistent with the representation that the analogical reasoner can use. Each scenario which is stored is independent of the others.

Usually, there is a one-one or one-many correspondence between a stored situation and an operator schemata in the planning knowledge-base. Given the correspondence, we can transform a stored situation into a planning operator/action and vice versa. For instance, look at figure 4.7. Of course, the figure shows the graphical representation of the situation. The actual representation for these scenarios is frame-based. Appendix A gives the actual representation of some of the scenarios.

The situation repository contains not only information about various situations encountered in the past, it also consists of frame-base knowledge representations of some of the specific objects in the domain like the sword and the axe. This knowledge is required for improvising based on *precondition satisficing* which is discussed in the next chapter. Figure 4.12 shows the actual representations for some of the objects.



```
% SWORD
?- defconcept(sword, [
        [isa, 0.5, weapon],
        [action, 0.5, thrust],
        [attributes, 0.5, long, sharp],
        [purpose, 0.5, kill_victim]]).
% AXE
?- defconcept(axe, [
        [isa, 0.5, weapon],
        [action, 0.5, chop],
        [attributes, 0.5, sharp],
        [purpose, 0.5, chop_tree, kill_victim]]).

%WEAPON
?- defconcept(weapon, [
    [isa,    0.5,  instrument],
    [attributes,    0.5,  dangerous],
    [purpose, 0.5, attack, self_defense, tool]]).
```

Figure 4.12: Actual representations of objects in situation repository

Again, the situations in the situation-repository can be knowledge-engineered or can be learnt incrementally. But there should be at least one situation in the repository for the analogical reasoner to work. We have knowledge-engineered a few situations to provide the initial set. We could also set up a feedback loop that will create and add new situations to the repository from the newly formulated actions. Remember that in the course of new operator/action formulation, we retrieve and adapt an old situation, say S_1, to get a new situation, say S_n. From S_n, we derive a new operator/action. The only thing that keeps us away from storing the situation S_n among other situations in the repository is the correctness aspect. We do not know if the new action/operator derived from S_n is correct or not until it has been included in some plan and that plan has been executed successfully. With a feedback loop in place, we can evaluate the correctness of the newly formulated operator/action based on the successful execution of the plan or not. If the newly formulated operator/action was correct, then S_n, from which the new operator/action was derived, can be added to the situation repository. Thus with a small feedback loop in place, the situation repository can grow incrementally.

4.9 Summary

In this chapter we took a reductionist approach and looked at the various components that constitute the architecture of CREAP. We saw that CREAP's architecture consists of two main modules, the planning module and the improvisation module. We also saw that the planning module consists of a classical partial-order-planner (POP). This POP uses two container components, the operator model and the domain model, in order to synthesize plans. This planning module is more or less similar to any standard classical planning system. However, what is novel to this architecture is the improvisation module that operates in conjunction with the planning module. We have shown that this improvisation module is responsible for taking care of situations where the planning module fails. The improvisation component consists of the current-situation generator, the analogical reasoner and the situation repository which is a container component.

In the next chapter we take a holistic view of CREAP's architecture and explain how all of the components work together to produce the behavior that we are seeking.

Chapter 5

Planning and Improvisation Process in CREAP

In this chapter we will look at CREAP's architecture as a whole and emphasize the process that is involved in extracting the improvisational behavior. In the previous chapter we studied the different components piece-by-piece. As such, some of the issues involving the interaction of these components may have remained obscure. We hope to clear that in this chapter.

In chapter 4 we looked at the various pieces that make-up CREAP's architecture and we discussed the function of each of those pieces in detail. In this chapter we examine how they all come together to address the problem of plan improvisation in CREAP.

The behavior of CREAP arises from the interaction of three distinct processes planning, plan execution and plan improvisation, all connected in a *plan-execute-improvise cycle.*

The cycle of planning, execution and improvising

Before CREAP goes live, the following have already been done: CREAP's world has been designed, CREAP has been programmed with the goal it is supposed to achieve, and CREAP's domain model, operator model and situation repository are in place.

When CREAP goes live, it starts with the planning process. CREAP's *planner* receives the goal (which has been programmed into CREAP). The planner attempts to produce a high-level plan using the existing *operator model* and *domain model.* This can result in two cases: (1) the planner fails to synthesize a high-level plan, and

(2) the planner is able to synthesize a high-level plan.

Case 1: If the planner is NOT able to produce the high-level plan, this makes a case for the improvisation process to try to do something. Section 5.1, *improvising from planning failure*, talks about this case.

Case 2: Assuming that the planner is able to produce a high-level plan, the plan is passed on to the *executor*. The executor's basic function is to accept the high-level plan and translate it into a set of primitive executable moves, and then execute those moves as described in section 4.2. This may again lead to two cases: (1) the executor succeeds resulting in the achievement of CREAP's goal, or (2) the executor fails due to an incorrect domain model, allowing the improvisation process to takeover. What is interesting here is case 2, which is the interaction between the execution process and the improvisation process. Section 5.2, *improvising from execution failure*, talks about this case.

Figure 5.1 sketches the overall planning-execution-improvisation process.

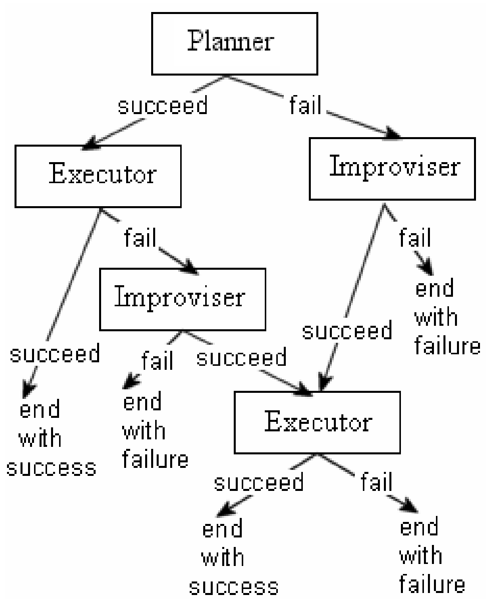


Figure 5.1: Overall planning-execution-improvisation process

5.1 Case 1: Improvising from planning failure

In this section we are interested in situations when, once a goal is given to CREAP, the planner fails to create a plan. We had earlier discussed that such a situation can arise if the operator model is incomplete. One way to try and salvage this situation is to formulate a new operator which will fill the hole in the operator model. We shall see in the course of this section how analogical reasoning helps in *new operator formulation* and also some of the issues related with this approach. Here is an overview of improvising from planning failure.

- CREAP's operator model is incomplete and as a result the planner fails.
- The improvisation process begins. The current-situation-generator looks at the failed situation, makes a report, and passes it to the analogical reasoner.
- The analogical reasoner tries to retrieve a case from the situation-repository which could be helpful in improvising in the present failed situation.
- Using the information from the retrieved case, the analogical reasoner tries to formulate a new operator that might work in the failed situation, allowing the planning to go through.
- If a new operator is formulated, the control goes back to the planning process and the new operator is added to the operator model. The planner resumes and completes the generation of a high-level plan.
- This high-level plan is then given to the executor to execute. If the execution succeeds, the operator formulated was correct. It is retained in the operator model. Else, it is removed from the operator model.

We shall examine the above process with the help of a hypothetical planning exercise. Let us consider a domain which is different from CREAP's own domain.

Let the domain involve an agent planning to build and move into a new house. Refer to figure 5.2. Let us assume that the agent's planner is given the goal `in(own_house)` to satisfy, as illustrated in figure 5.2.

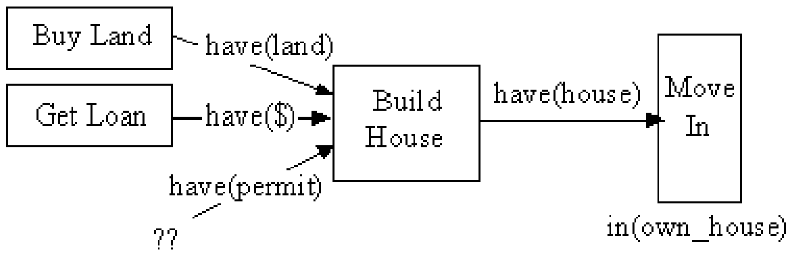


Figure 5.2: Situation of a plan failure due to an incomplete operator model

As we know, the planner works backward from the goal. The planner has started from the `in(own_house)` goal and reaches a point in the planning process where it is examining the operator `Build House` operator as shown in figure 5.2.

The no-permit-situation

The `Build House` operator has three preconditions to satisfy: `have(land)`, `have($)` and `have(permit)`. Further, let us assume that it has `Buy Land` and `Get Loan` operators in the operator model, which satisfy the preconditions `have(land)` and `have($)`. Assume that there is no action that will satisfy the precondition `have(permit)`. This means that the planning agent in this situation does not know how to get the permit. As a result the planner fails to construct a plan. This is the end of the road for most classical planners. Let us refer to this failed situation as the no-permit-situation. At this stage figure 5.3 tells us where we are in the overall planning-execution-improvisation process.

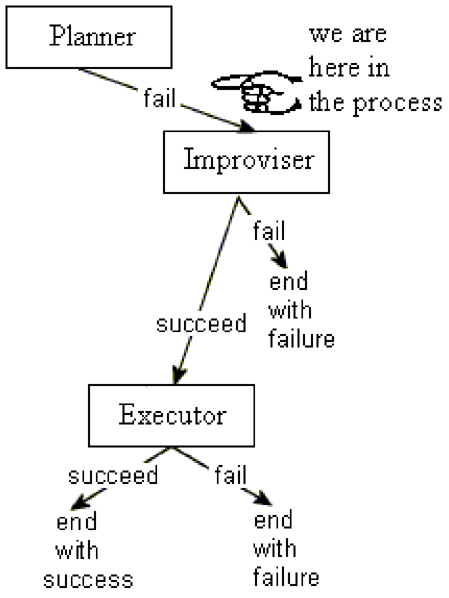
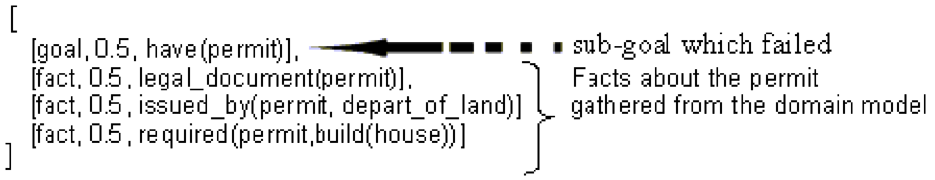


Figure 5.3: Stage in the overall process after the planner fails

Let us see how our analogy-based improvisation process can help in situations such as the no-permit-situation. When the planner fails, the improvisation process begins. It starts with the *current-situation generator* preparing the current-situation *report*. The details of how this component works can be found in the previous chapter.



```
[
  [goal, 0.5, have(permit)],
  [fact, 0.5 , legal_document(permit)],
  [fact, 0.5 , issued_by(permit, depart_of_land)]
  [fact, 0.5 , required(permit,build(house))]
]
```

sub-goal which failed
Facts about the permit gathered from the domain model

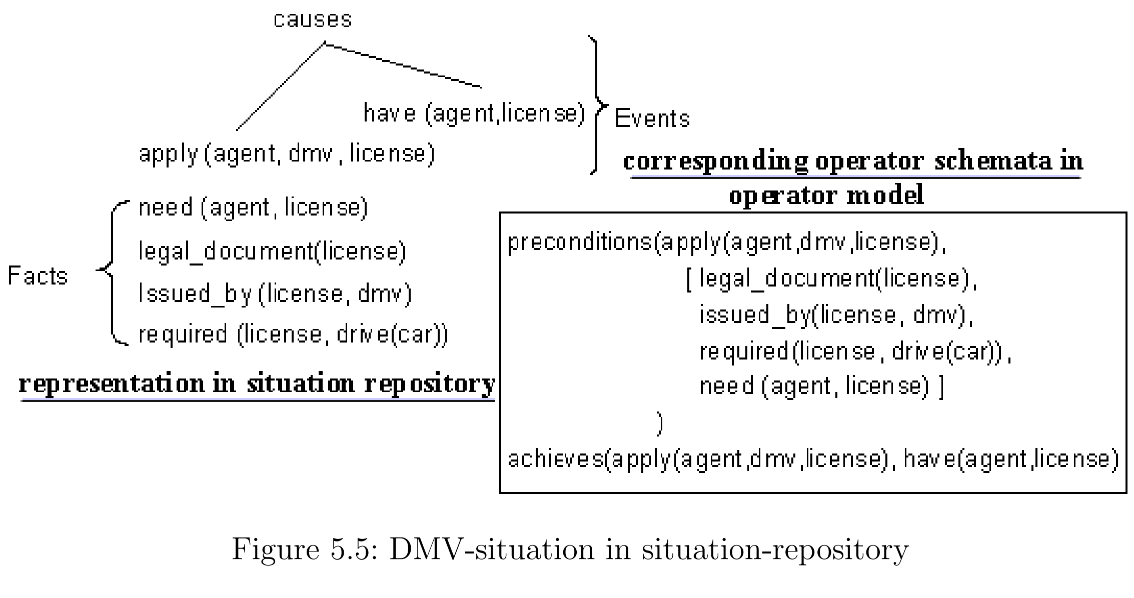
Figure 5.4: A report generated by the current-situation generator

Let us assume that based on (1) the goal that the agent was trying to solve when the planner failed and (2) the domain information, the current-situation generator generates the report shown in figure 5.4.

Now this current-situation report is passed to the *analogical reasoner* of the improvisation module. The function of the analogical reasoner is to retrieve situations analogous to the no-permit-situation (where the planner failed) from its situation repository (memory). The success or the failure of the improvisation process depends on whether an analogous situation exists in the memory or not, and also on the analogical reasoners ability to see the analogy.

The DMV-situation

For the sake of demonstration let us assume that in the recent past the agent has bought a new car and had to apply for a license to drive it. Without the license the agent would not have been able to drive the car. So the agent had been to the DMV (department of motor vehicles) and applied for one. Let us refer to this situation as DMV-situation. This episode was stored in the agent's memory as in figure 5.5.


causes
have (agent,license)
apply (agent, dmv, license)
Events
corresponding operator schemata in operator model
Facts
need (agent, license)
legal_document(license)
Issued_by (license, dmv)
required (license, drive(car))
representation in situation repository

```
preconditions(apply(agent,dmv,license),
                [ legal_document(license),
                  issued_by(license, dmv),
                  required(license, drive(car)),
                  need (agent, license) ]
             )
achieves(apply(agent,dmv,license), have(agent,license))
```

Figure 5.5: DMV-situation in situation-repository

If the situation-repository does contain such a situation, the SAPPER-based analogical reasoner can see the structural similarity between the two situations (the no-permit-situation and the DMV-situation) and this yields a structure-mapping which can be used in the no-permit-situation to improvise the plan. Based on this structure-mapping we can infer new knowledge about the no-permit-situation (see figure 5.6).

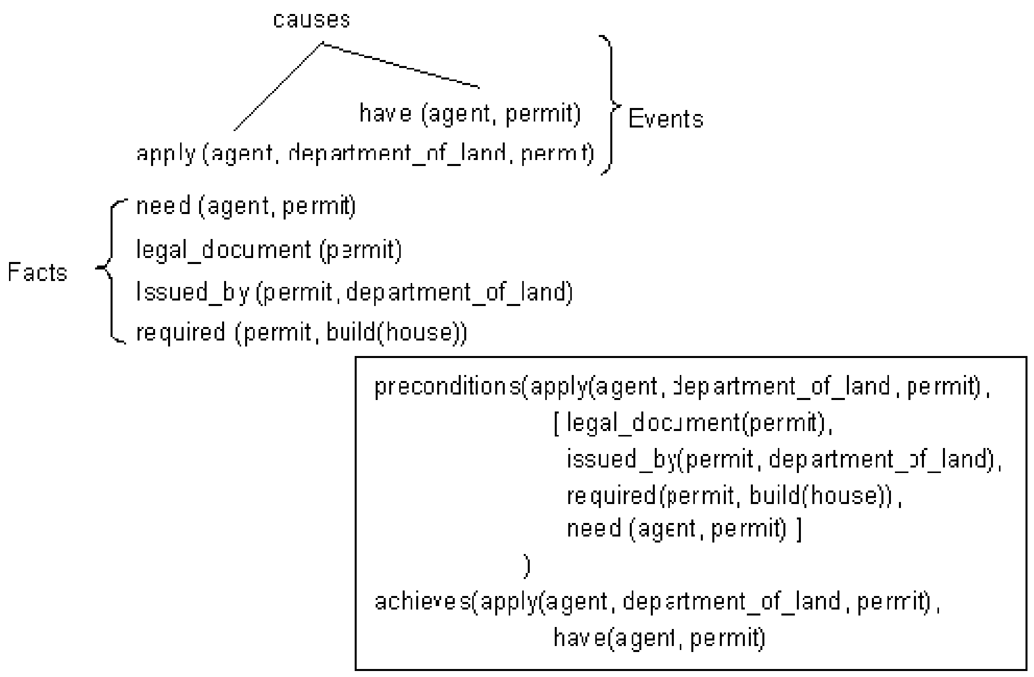

causes
have (agent, permit)
apply (agent, department_of_land, permit)
Events
Facts
need (agent, permit)
legal_document (permit)
Issued_by (permit, department_of_land)
required (permit, build(house))
preconditions(apply(agent, department_of_land, permit),
[legal_document(permit),
issued_by(permit, department_of_land),
required(permit, build(house)),
need (agent, permit)]
)
achieves(apply(agent, department_of_land, permit),
have(agent, permit)

Figure 5.6: Newly formulated operator

The analogical inference results in a causal mapping and a set of facts in the no-permit-situation. Causal mapping leads to `apply(agent,department_of_land, permit)`, a new operator that was formulated based on the DMV-situation. The following facts associated with this causal relationship become the pre-conditions for the action: `need(agent, permit)`, `legal_document(permit)`, `issued_by(permit, department_of_land)`, `required(permit,build(house))`. It is reasonable to assume that these facts are either true in the domain (hence in the domain model) or there exist other actions that satisfy these pre-conditions.

The bottom-line is that we have taken an action from a different context and applied it to the present context. Here is an important thing to note: we have not formulated this new action in any combinatorial investigative fashion, nor have we used any context-independent search techniques. This approach to using analogical-reasoning is more cognitively plausible.

What do we do with this newly formulated operator? We include it in the partial solution (partial-plan) and also add it to the operator model with a temporary status. If we are able to arrive to this step, we are at a stage in the overall planning-execution-improvisation process indicated by figures 5.7.

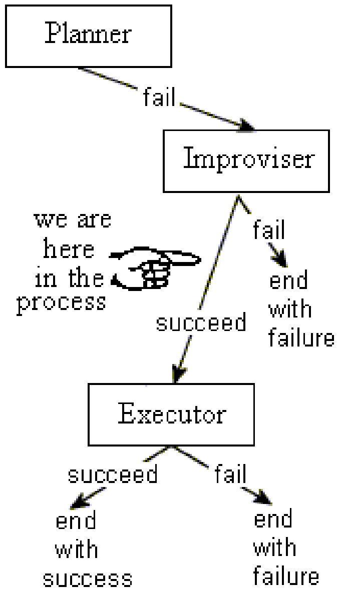


Figure 5.7: Stage in the overall process after the operator formulation

With the inclusion of the new operator in the partial plan, the job of the improvisation component is over. The planning module then resumes the planning process from where it had stopped. This time, with the new operator in the partial plan, the plan will mostly come through, assuming that there are no other holes in the operator model. There are a few issues that arise when we think of this approach.

Correctness of the new operator: *How do we know that the new operator that was formulated is correct?* There is no way to prove the correctness of the new operator prior to execution. One can never say a priori that the new operator will succeed in the external world. The only way to ascertain its correctness is to execute the solution containing this operator and monitor the execution. For instance, we will never be able to tell beforehand that applying for a permit will fetch the agent a permit. It can only be found out by actually trying to apply. If the agent was lucky enough to get the permit, the execution succeeded and the goal was achieved. Now we have moved to a stage indicated by figure 5.8(a). The other possibility is that when the plan is executed, the newly formulated operator was incorrect and it failed in the actual world, which takes us to a stage indicated by figure 5.8(b).

Growth of knowledge: *How does CREAP's knowledge grow as a result of improvisation?* We had mentioned that the newly formulated operator is added to the operator model with a temporary stamp. During plan execution if the new operator succeeds, it can be retained in the operator model on a permanent basis. However if the operator fails, we know that analogy did not work and the operator is removed. So the key to knowledge growth in CREAP is for the newly formulated operator to succeed during execution.

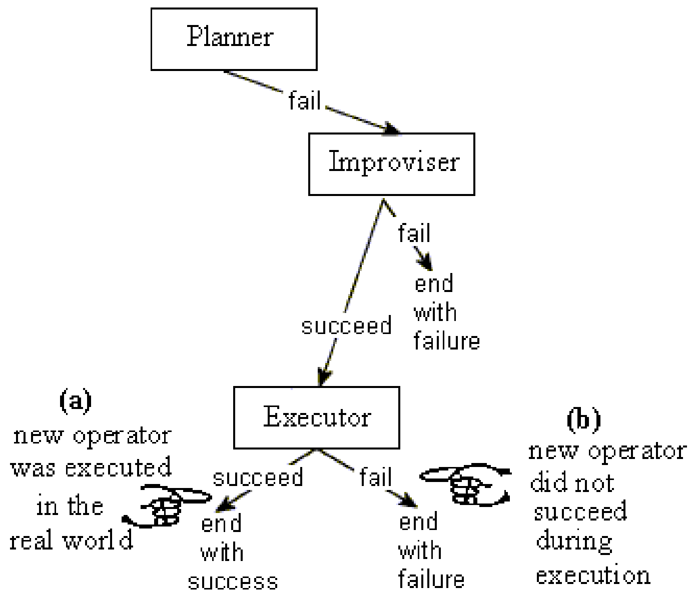


Figure 5.8: Stage in the overall process as a result of execution of improvised plan

So far in this section we looked at Case 1. Here when CREAP goes live, the operator model is incomplete and the planner is unable to synthesize a plan to begin with. Then we saw how the improvisation process interacts with the planning process to formulate a new operator that fills the gap in the operator model and helps in completing the planning and execution process.

5.2 Case 2: Improvising from execution failure

In the previous section we talked about improvising from planning failure. In this section we assume that the planning goes through resulting in the creation of a high-level plan. However, we are interested in siuations when the execution of this plan fails, which is Case 2.

The earlier case was different because the planner was unable to create a plan in the first place. Here we have a plan, but the execution fails. Such a situation arises because of the incorrectness problem. The incorrectness problem arises when a certain fact is true in the domain model but not so in the real world, and the planner uses this fact in constructing the plan.

When a failure occurs during planning (Case 1), we used the analogy-based operator formulation to improvise. When a failure occurs during plan execution (Case 2), we can improvise using two methods: *precondition satisficing* and *operator formulation* (which is the same as in the previous case). We shall be discussing these as we go along. But first let us examine Case 2 with yet another agent, planning a dinner for a few guests.

The dinner planning agent

Let us assume that the agent does create a high-level plan for the dinner from `Start` to `Finish`, containing an action `open(wine_bottle)` with the schemata:

```
preconditions(open(wine_bottle), [have(cork_opener)])
achieves(open(wine_bottle), pourable(wine))
```

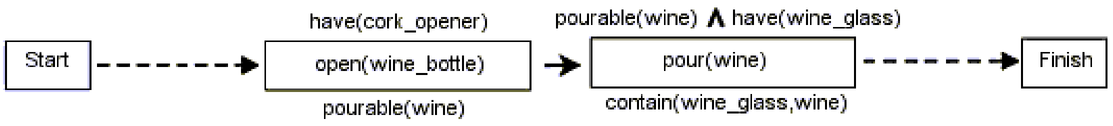


Figure 5.9: Dinner plan from start to finish

During execution, when the agent reaches the `open(wine_bottle)` step it notices that the cork opener is unusable due to wear and tear. The execution of the initial plan fails. Since the execution is being monitored, the agent becomes aware of the execution failure. This takes us to a stage shown in figure 5.10 in the overall process.

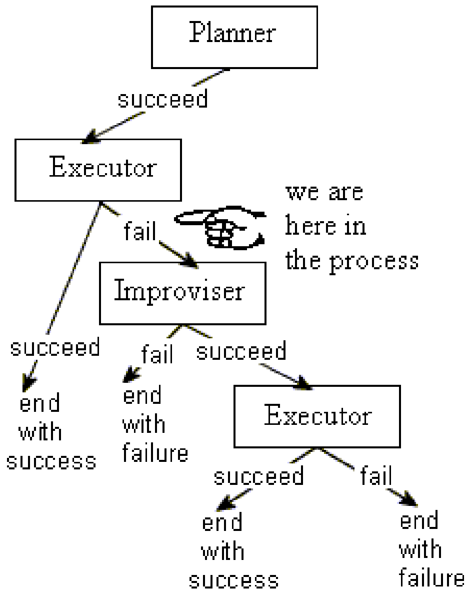


Figure 5.10: Stage in the overall process when execution fails

This failure exposes an incorrectness in the domain model of the agent. For this plan to have succeeded, the fact `have(cork_opener)` must have been true in the domain model of the agent. Else, this plan would not have materialized in the first place. But only during execution does the agent find that the cork opener is unusable.

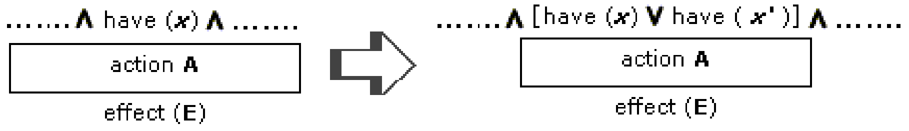
This negates the `have(cork_opener)` fact, hence the incorrectness in the domain model.

In such situations the natural reaction of people would be to improvise on the plan by acting on the `open(wine_bottle)` action with another object that is comparable to the cork opener. For instance they may try to remove the cork with a long screw and pliers. This is similar to a person using a kitchen knife to sharpen her pencil when she cannot find a pencil sharpener. We refer to this behavior as "precondition satisficing." So in precondition satisficing, if an object x in the precondition of an action A is responsible for the failure of A, we look for another object x' comparable to x, and try A with x'. The analogy comes into play when we are trying to look for x' which is similar to x.

If precondition satisficing also fails, there is another way to improvise - *operator formulation*, which we discussed in the last section. Suppose action A fails, and A had the effect e. Here we are not concerned with the preconditions, but the effect e of of A. If A fails, we create an altogether new action A' having the same effect e and adjust the original plan by including A'. We shall discuss both these methods in the following sections.

5.2.1 Precondition Satisficing

In precondition satisficing, the agent is executing a plan when one of the actions fails because one or more preconditions (which were thought to be true at the time of planning) are not satisfied. Under such circumstances, if the failed precondition involves an object x, the agent looks for similar object(s) x' which can be substituted for x, thus trying to satisfy the failed precondition. Figure 5.11(a) shows an abstract operator schema before and after precondition satisficing. Note that the precondition $\texttt{have}(x)$ is modified to $[\texttt{have}(x) \bigvee \texttt{have}(x')]$. This adjustment to the precondition using disjunction, to take into account the incorrect domain model, allows the planner to go through with the plan. Figure 5.11(b) shows the schema change in the `open(wine_bottle)` action.



.......∧ have (x) ∧ ....
action A
effect (E)
.......∧ [have (x) ∨ have (x')] ∧ ....
action A
effect (E)

Figure 5.11a: Precondition satisficing (abstract schemata)

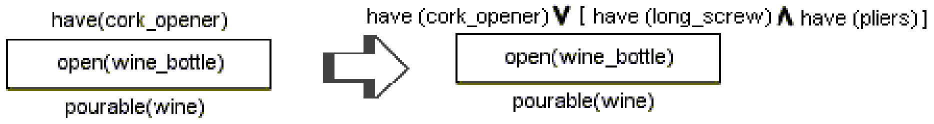


Figure 5.11b: Precondition satisficing in the open(wine_bottle) operator

This sort of precondition satisficing could lead to a plan improvisation of the original dinner plan as shown in figure 5.12.

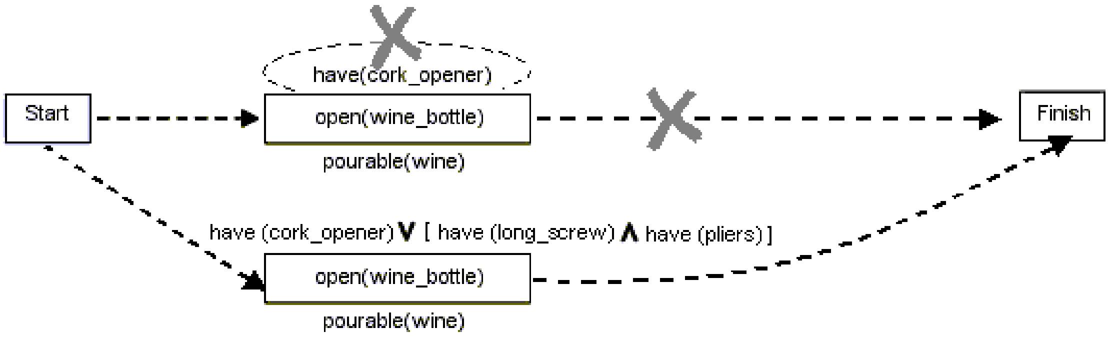


Figure 5.12: Plan improvisation by precondition satisficing

The key to this sort of precondition satisficing improvisation involves seeing the analogy between the two objects in question and revising the operator using disjunction. In this case it is the analogy between the *cork opener* and the *long screw* and *pliers*, and using the result of this analogical inference to modify the failed action `open(wine_bottle)`. If the architecture allows the agent to employ such a mechanism, it can sometimes overcome difficulties in planning arising due to a dynamic world. But the agent should have *sufficient knowledge about the objects*, at least their features. For the agent in the above example the *cork opener* and the *long screw* were not mere symbols. There was more information associated with these objects that allowed the agent to see the similarity between the two. If when confronted with $AA : BB :: MM :?$, we readily recognize that ? corresponds to NN, then A, B, M and N are not mere symbols. We know more about these symbols, the relationships between them, which lets us see the similarity.

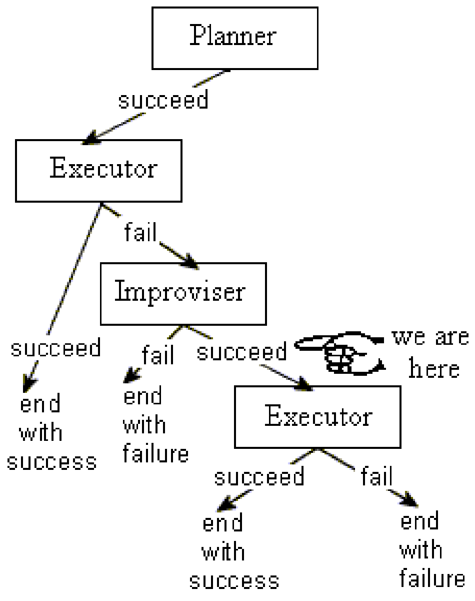


Figure 5.13: Stage in the overall process after precondition satisficing

Assuming that the improviser is successful at precondition satisficing, figure 5.13 tells us where we are in the overall process. The planner succeedes in creating the initial plan, but it failed during execution. Then the improviser succeeded by means of preconditioning satisficing. As a result, we have an improvised plan.

This improvised plan, as customary, is not guaranteed to succeed. If the executor succeeds, the plan was right and the goal is reached, which is the end of the overall process. On the other hand, the improvised plan could also fail during execution. Assume that when the agent tried the obvious alternative of using a long screw and pliers, the screw did not adequately bite-into the cork and upon pulling, it came free. However the cork remained inside the bottle. Should the agent give-up if the improvisation process fails once? This need not be so. The agent can try any number of alternative solutions to the failed precondition. But there is a limit to precondition satisficing. Sometimes we run out of things that we can use in failed situations. Sometimes problems require novel solutions, a totally different way of thinking about the problem and solving it. That is why operator formulation is also necessary.

5.2.2 New operator formulation

So far, the improvisation we talked about involved using the failed action in the plan and adjusting it in many ways by means of precondition satisficing. But sometimes we will have to formulate entirely new action(s) which will produce the same intended effect as the failed operator. For example, if a household cleaner fails to remove mineralization on a faucet, precondition satisficing suggests trying another cleaning solution, say solution X. If solution X fails too, precondition satisficing suggests yet another cleaner, say solution Y. This can go on. But if none of the cleaning solutions available are able to remove the mineralization, one might have to think of a totally new action like abrading it using a coarse material. The agent will have to *formulate* an operator with the same intended effect as the failed operator. We refer to this type of improvisation as *operator formulation*.

Let us go back to planning a dinner example. When the agent tried to remove the cork using a screw and failed, it cannot continue precondition satisficing as it cannot "think" of any more substitutes for the cork opener. Therefore the agent has to formulate a new action which has the same intended effect of `pourable(wine)` as the `open(wine_bottle)` operator. This is similar to the operator formulation we discussed in section 5.1.1, improvising from planning failure - Case 1.

To achieve the operator formulation, the agent has to be reminded of a relevant situation in the past. If such a scenario exists, the information exchange between the past scenario and the current failed situation helps in creation of a new operator for the current situation. The inclusion of this new operator in the plan marks the improvisational behavior that we are looking for. In the dinner example, let us assume that the agent is reminded of a situation when it had forgetfully placed a bottle of soda in the deep-freezer. The soda had solidified inside the bottle and the bottle had cracked. Based on this, the agent tries to separate the wine from the

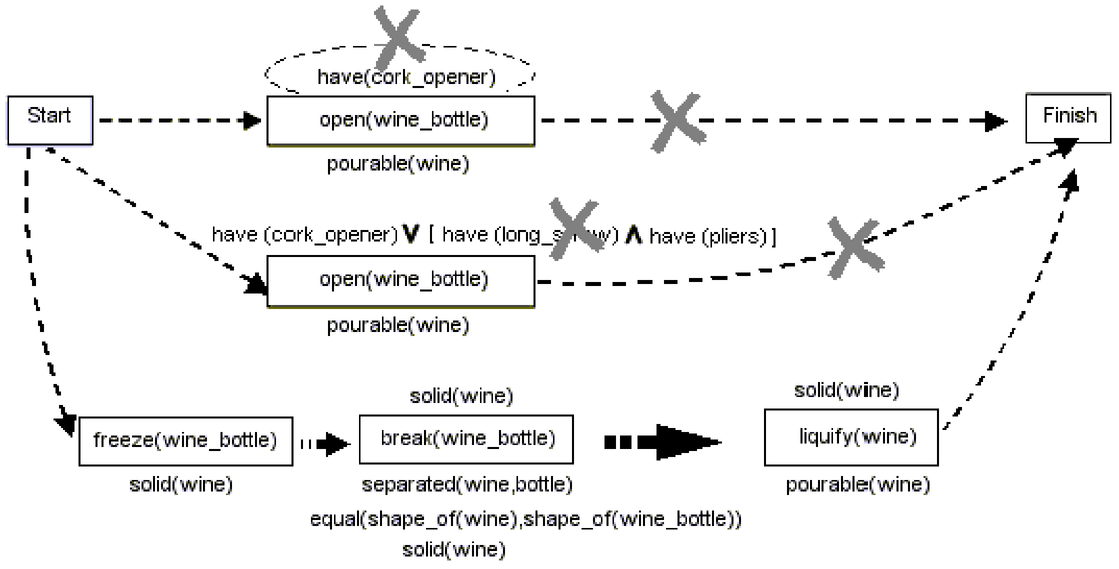


Figure 5.14: Stage in the overall process after precondition satisficing

bottle by freezing it, breaking the bottle to separate the wine, and liquefying the wine in a different container. Although as obtuse as it may sound, it is a legitimate solution to the problem, and one that may be considered creative in some sense.

Here we see an improvisation that discards the action in the original plan and formulates a new one with the same effect, as shown in figure 5.14. This is the primary difference between the two modes of improvisation: in precondition satisficing, the failed operator is adjusted till the failed precondition is or can be satisfied. When one runs-out of objects that can be used for precondition satisficing, one has to discard the original operator and try to formulate a new one having the same intended effect as the original operator in the plan, and which will have a different set of preconditions.

Although both precondition satisficing and operator formulation use a notion of analogy, the other important difference is that in precondition satisficing the analogy sought for is usually an object and is based on feature-wise or surface similarity. On the other hand, in operator formulation, the analogy sought for is a situation and is usually based on the structural similarity of the two situations (similarity at the level of relationships, rather than individual features). Because of the deep analogy involved in the operator formulation improvisation, it gives an impression that the solution to the planning failure problem is creative or novel.

5.2.3 Precondition satisficing takes precedence

In the previous section we saw that when a failure occurs at the plan execution stage, improvisation can take place either using precondition satisficing or operator formulation. In this section we will argue that precedence must be given to precondition satisficing. If this fails, only then we should consider the new operator formulation. The reason is as follows.

The analogy-making process can be straightforward or complex. An example of a straightforward analogy process is when a student takes the example worked out in a section of a mathematics text and maps it into the solution for a problem in the exercises at the end of the section. An example of more complex analogy-process is when Rutherford used the solar system as a model for the structure of the atom in which electrons revolved around the nucleus in the way that the planets revolved around the sun [12].

So what distinguishes between a simple analogy and a hard/deep analogy? In a simple analogy, people spontaneously perceive the relevance of an object or a situation to the current problem. In hard or deep analogy, most people do not readily notice the relevance of one situation to another. They have to be given clues that promote the transfer of knowledge from one situation to another. Only then will they notice the similarity. Gick and Holyoak did an experiment in which they read to subjects the *general and the director* story and then *Duncker's ray problem*, which were analogous [3]. Very few subjects spontaneously noticed the relevance of the first story to solving the second. To achieve success, subjects had to be explicitly told to use the general and director story as an analogy for solving the ray problem.

When subjects do spontaneously use previous examples to solve a problem (simple analogy), they are often guided by superficial similarities in their choice of examples. For instance, Ross [28] taught subjects several methods for solving probability problems. These methods were taught with respect to specific problems. Subjects were then tested with new problems. It turned out that subjects were able to solve more problems that were superficially similar to the prior examples (same principles were required to solve the example problems and the test problem).

So, this suggests that people have an innate predisposition towards perceiving superficial similarity. This may be because deep analogy involves expending more cognitive resources. Among the two methods of improvisation that we have seen in this section, precondition satisficing is closer to the simple analogy-process and operator formulation is closer to the deep-analogy process. Therefore, we argue that for cognitively plausible models of analogy-based improvisation, precedence has to be given to precondition satisficing, where we look for *superficial feature-based* similar examples to satisfy the preconditions of a failed operator. Only if precondition satisficing fails do we get into a *deeper relations-based* operator formulation method.

5.2.4 Summary

In this chapter we saw how incompleteness in the operator model and incorrectness in the domain model can lead to failure at the planning stage and the execution stage respectively. Improvisation at the planning failure stage involves new operator formulation. Improvisation at plan execution can take place either using precondition satisficing or operator formulation. We examined both these methods for improvisation in detail and also saw how improvisation by either of these methods could potentially lead to the success of the overall planning process. We have also argued that precondition satisficing takes precedence over operator formulation.

Chapter 6

Experimental observations

This chapter presents four experiments which are examples of CREAP engaged in improvised behavior. These experiments highlight the architecture of CREAP, demonstrating its ability to reproduce and deal with characteristics of improvisational behavior in planning.

The validity of any scientific model hinges on its ability to accurately represent the behavior which it was designed to explain. In proposing a new model, there are two requirements that arise from this: to illustrate that the model does indeed explain the behavior on which it is based, and that it does so more accurately or extensively than existing models. In the case of CREAP architecture, these requirements have to a large extent already been met. Chapter 2 has outlined the limitations of the classical planning techniques in general and with respect to the phenomena displayed during the performance of improvisation. Chapters 4 and 5 have presented an approach and an agent architecture based specifically on the phenomena of improvisational behavior. In particular, we have shown that conceptually the CREAP architecture both subsumes the capabilities of a classical planner and explains far more complex phenomena: how an agent can go beyond specific and deliberative planning, making use of deeper world knowledge when necessary, allowing the agent to perform more flexibly and innovatively.

The fact that the CREAP architecture and the improvisational behavior it embodies can be used to explain these phenomena validates the approach in part. In AI, however, we have the ability to do much more than illustrate how well new theories work on paper: we can make the theory active through a computational implementation, physically demonstrating both its advantages and its limitations. In order to further demonstrate and validate both the improvisational approach and the CREAP architecture, and to serve as a basis for future research, this Chapter describes a modest implementation of CREAP engaged in episodes of planning activity in a simulated environment of its own.

6.1 Scope of experimentation

The implementation of CREAP was in LPA Prolog, using the V-World environment described in chapter 2. The features provided by V-World present a natural fit for experimentation with CREAP's architecture. A major difficulty in experimentation with CREAP is the scope of such experimentation. Building a complete improvising agent involves solving a great many open problems in AI. The implementation described here emphasizes the decision-making components of improvising agents at the expense of aspects of physical embodiment, such as completely realistic vision or manipulation. The facilities provided by V-World for obtaining sensory information and manifesting the effects of the actions are employed to substitute for these features.

This experimentation is not only limited by technological resources necessary to build a full CREAP architecture, however. As chapters 4 and 5 have illustrated, the architecture is also limited by the knowledge requirements. The extent of the domain and operator models and also the collection of background knowledge in the situation repository determine, to a large extent, the ability to plan and improvise. While one can expect even relatively undersized operator and domain models to cover most of the planning situations in restricted domains, the collection of background knowledge that is required for even mundane improvisations is vast and would require an extensive effort to even begin to deal with realistically in the time-span available for the research described in this thesis. To provide a basis for comparison, the CYC project, implementing background knowledge available to an average six-year-old, estimates a two person-century effort [15]. So, acquiring domain and operator models required for planning is relatively insignificant when compared with the issue of acquiring the background knowledge required for improvisation.

Because of the difficulties of scope on the resource-bounds of this project, the extent of this CREAP's implementation and this experimentation itself is necessarily limited. We have already discussed the implementation in the earlier chapters. In this chapter we shall look at the limited experimental set-up and the results of the experiments.

6.2 The Experimental environment

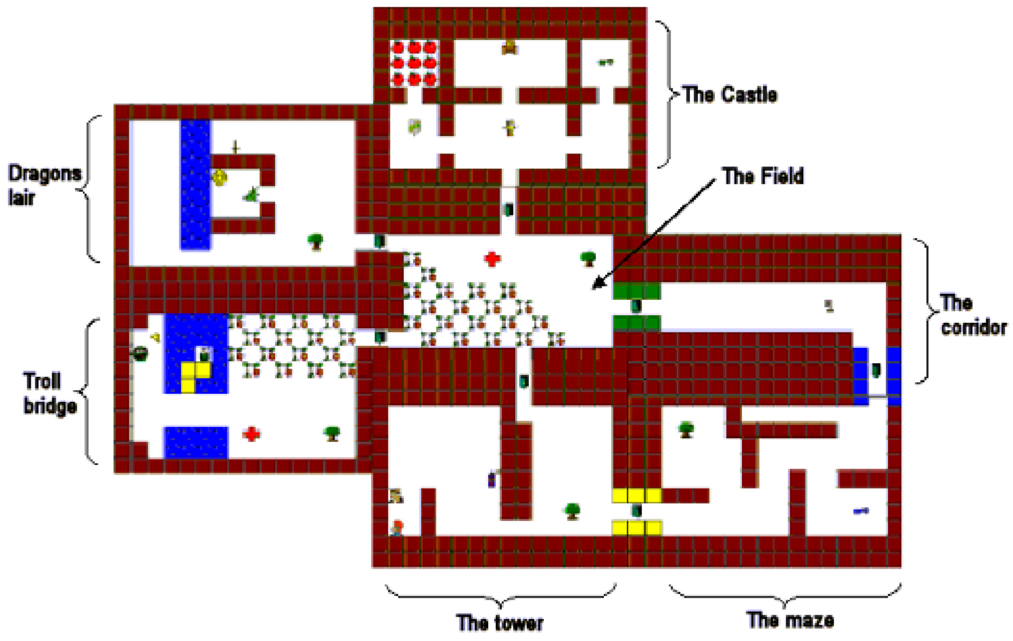


Figure 6.1: Experimental environment

We have already seen the experimental environment when we discussed the CREAP's world in Chapter 3. Figure 6.1 shows a graphical view of the environment and section 3.1.1 is devoted to explaining the features of the same.

Further, CREAP is the protagonist. Jane is a princess who is been captured by the wicked hag and Jane is under the spell of this hag. The hag wants the gold and the bird in order to uncast the spell and free the princess. Meanwhile, the gold is guarded by a dragon and the bird is guarded by a troll. Both the troll and the dragon are dangerous and could potentially kill CREAP. There are various other obstacles that CREAP can face in this world like hornets, thorns, etc.

The goal for CREAP is to plan so that it will rescue Jane from the hag and return her to the throne, making sure that it doesn't get itself killed in the process. Once CREAP has rescued Jane from the hag, it has to prepare a safe path which Jane can follow. Jane is very fragile character. If the path is not safe and if Jane comes in contact with other harmful entities, she gets killed.

6.3 Knowledge structuring

As an autonomous, resource-bound agent, CREAP operating within the environment described above has its own limited knowledge of the world that it inhabits. As we have mentioned earlier, CREAP's knowledge of the world is distributed among three container components: the domain model (which consists of facts that are **supposedly** true in the world), the operator model (which consists of actions that CREAP can perform in its world,) and the situation-repository/episodic-memory/background-knowledge (which contains situations or scenarios that CREAP has encountered in the past).

The initial knowledge in these three structures is knowledge engineered. Since CREAP's world is a restricted domain, the domain and the operator models can be engineered to be complete with respect to CREAP's goal. That is, there is a universal set of actions and facts about the world, which when embodied by CREAP allows it to plan and execute without failure, resulting in the achievement of its goal. That *universal set of operators* or the complete domain model consists of the following actions (represented in STRIPS-style) shown below:

```
Universal set of Operators
---------------------------
% take_to_throne
preconditions(take_to_throne(agent,jane),
     [near(agent,jane),
      free(jane)])
achieves(take_to_throne(agent,jane), on_throne(jane)).

% bribe_hag
preconditions(offer(agent,hag,and(gold,bird)),
     [near(agent,hag),
      guards(hag,jane),
      goes_away_with(hag,and(gold,bird)),
      has(agent,bird),
      has(agent,gold)]).
achieves(offer(agent,hag,and(gold, bird)), free(jane)).
deletes(offer(agent,hag,and(gold,bird)), near(agent,hag)).

% strike_troll
preconditions(strike(agent,troll,axe),
     [near(agent,troll),
      has(agent,axe),
      guards(troll,bird),
      goes_away_with(troll,axe)]).
achieves(strike(agent,troll,axe),free(bird)).
deletes(strike(agent,troll,axe),near(agent,troll)).

% strike_dragon
preconditions(strike(agent,dragon,sword),
     [near(agent,dragon),
      has(agent,sword),
      guards(dragon,gold),
      goes_away_with(dragon,sword)]).
achieves(strike(agent,dragon,sword),free(gold)).
deletes(strike(agent,dragon,sword),near(agent,dragon)).

% get_object
```

```
preconditions(get(agent,Object),
    [near(agent,Object),
     free(Object)]).
achieves(get(agent,Object),has(agent,Object)).
deletes(get(agent,Object),near(agent,Object)).

% goto
preconditions(goto(agent,Object),[ ]).
achieves(goto(agent,Object),near(agent,Object)).
```

Similar to the complete operator model, there exists a *necessary set of facts* that CREAP's domain model should contain. This set which CREAP should know and also which should be true for uninterrupted, first-time successful planning is shown below. These are the initial conditions in the world as we have designed it.

```
Necessary set of Facts
----------------------
    situation(guards(hag,jane), init).
    situation(guards(dragon,gold),init).
    situation(guards(troll, bird), init).
    situation(goes_away_with(hag, and(gold,bird)), init).
    situation(goes_away_with(dragon, sword), init).
    situation(goes_away_with(troll, axe), init).
    situation(free(sword), init).
    situation(free(axe), init).
    situation(requires(troll, food),init).
    situation(only_way_to_get(troll, food, bridge), init).
    situation(limited(trollfood), init).
    situation(depend(troll,bridge,trollfood), init).
```

Note that the facts that are mentioned above are the necessary set, but not the complete set. CREAP needs to know a lot more about the world and the other entities. But that knowledge is gathered via perception. For instance, when it perceives a troll, CREAP has a way of determining whether the troll is harmful or not. Therefore CREAP also gathers a lot of knowledge, on a need to know basis, as it performs in its world. The set of facts mentioned above is the knowledge required beyond the perceptual knowledge in order to complete the plan and execute it.

But if this is true, there is no scope for improvisation. We are in that idealistic setting which the classical planners assume. But in the course of experimentation, we will design experiments that will disturb this idealistic setting causing CREAP to fail at various stages. This failure will allow us to study its improvisational behavior.

6.4 Experimental set-up and observations

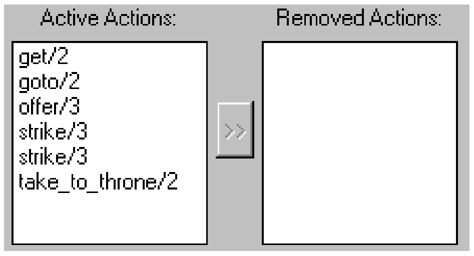
So, if CREAP's knowledge includes all the actions mentioned in the *universal operator set* and all the facts mentioned in *the necessary set of facts* mentioned above, CREAP can plan and execute to achieve its goal without any problem. This will be our first experiment, to make sure that CREAP achieves its goal in an idealistic setting.

However, that leaves no scope for improvisation. So, in experiment 2 we will remove an action that is necessary for CREAP to generate a plan. This makes the operator model incomplete, leading to failure at the planning stage. In a similar fashion, in experiment 3, we will change the world so that one of the facts mentioned in *the necessary set of facts* becomes invalid. This causes the domain model of CREAP to become incorrect, leading to a failure at the execution stage. This provides an opportunity for CREAP to improvise by precondition satisficing. In experiment 4, we will see that improvisation by precondition satisficing need not necessarily work, leading to new operator formulation, this time using deeper analogy.

Experiment 1

In experiment 1, we set up the world and CREAP in such a way that its domain model is sufficient and the operator model is complete and correct. Figure 6.2 shows a part of CREAP's interface which shows "Active actions" and "Removed actions."

Notice that all the actions are under "Active actions" and none under "Removed



get/2
goto/2
offer/3
strike/3
strike/3
take_to_throne/2

Figure 6.2: Experimental environment

actions." This means that all the actions in the operator model mentioned in the *universal set of operators* are intact and accessible to CREAP. Further we make sure that all the facts mentioned in *the necessary set of facts* are indeed true in the world. This is the initial condition set-up for experiment 1.

We begin the experiment by placing CREAP in its world and giving it the goal `on_throne(jane)` to achieve. This results in the successful plan shown below:

```
[
  init, goto(agent,sword), get(agent,sword), goto(agent,dragon),
  strike(agent,dragon,sword), goto(agent,gold), get(agent,gold),
  goto(agent,axe), get(agent,axe), goto(agent,troll),
  strike(agent,troll,axe), goto(agent,bird), get(agent,bird),
  goto(agent,hag), offer(agent,hag,and(gold,bird)),
  goto(agent,jane), take_to_throne(agent,jane), end
]
```

Now let us see the execution in example 1. Figure 6.3 shows the stages in the execution of the plan. As we can see the execution of the plan succeeds and CREAP is victorious in rescuing Jane and taking her to the throne, as expected.

The plan mentioned above is a high-level plan. Execution of this high-level plan involves a lot of complications. For instance, the high-level action `goto(agent,sword)` involves plotting a path from CREAP's current location to the location of the sword using a map and following the path. Meanwhile, CREAP should also keep track of its strength and damage. If either one of them is at an alarming level, CREAP must temporarily abandon its current course of action and seek out the nearest tree or cross. The other complication involves avoiding any potential threats like the hornet or the thorns.

What is shown is figure 6.3 are snapshots taken from the execution at moments when a particular high-level action is about to be accomplished. It should noted that there is no one-to-one correspondence between a high-level action and a snapshot in figure 6.3. For instance the second snapshot, where CREAP is near a sword is representative of two high-level actions in the plan: `goto(agent,sword)` and `get(agent,sword)`. Anyway, the bottom-line of experiment 1 is that CREAP was able to make a plan and execute it without any failures.

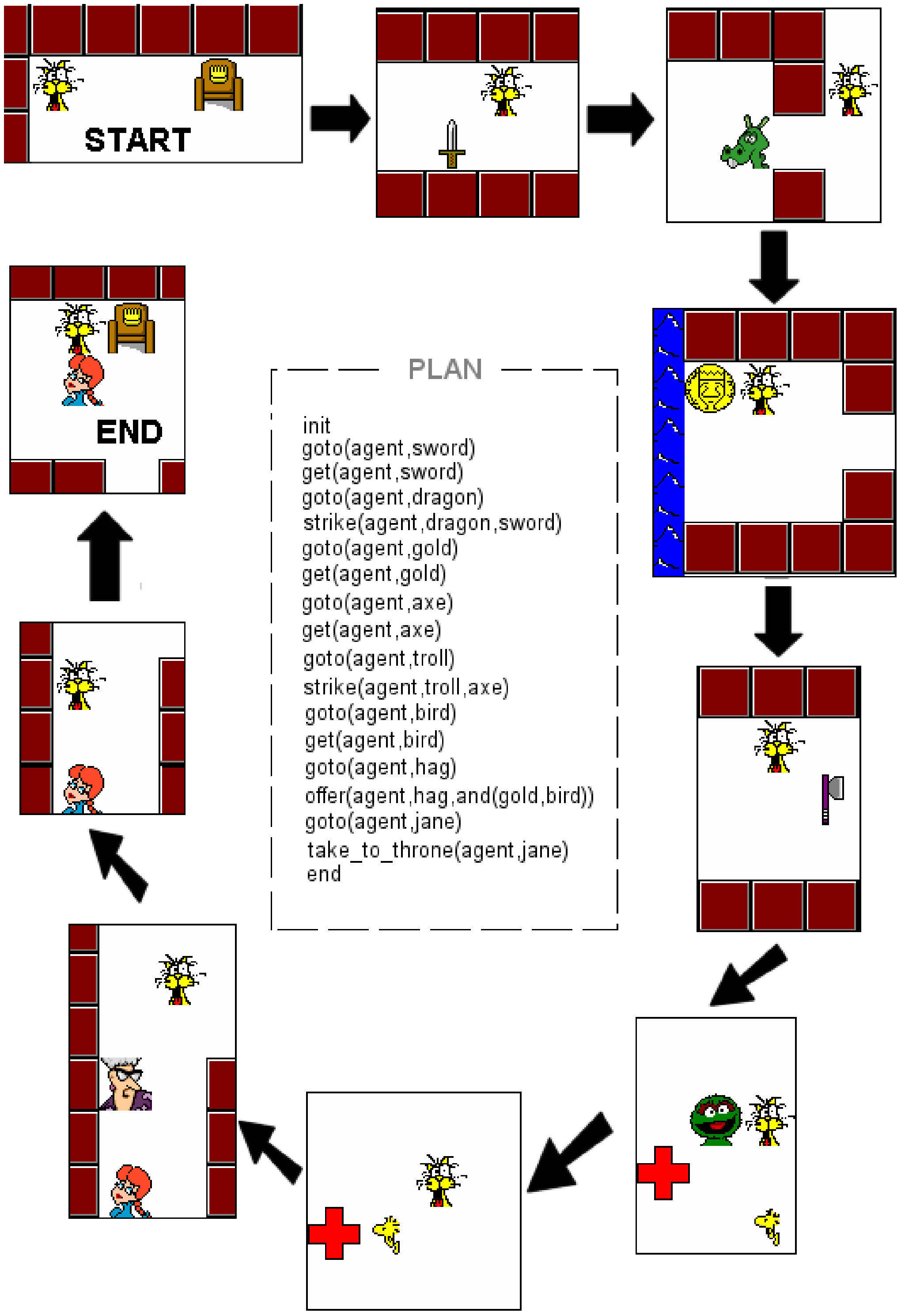
Experiment 2

In experiment 1 CREAP did not exhibit any improvisational behavior because it did not have to. All the knowledge required for planning was conveniently engineered. In experiment 2, however, CREAP gets the first opportunity to exhibit improvisational behavior.

In the set-up of this experiment, we remove a required action from the operator model. This leads to an incomplete operator model. Therefore CREAP fails to generate a high-level plan. The question of execution does not arise when we don't have a plan to begin with.

Figure 6.4(a) shows the actions that are still in the domain model and the one that has been removed. Note that the action `offer/3` has been removed from the domain model.

Now, when CREAP is asked to plan it fails with a message "Could not find a plan!" as shown in fig 6.4(b). This indicates a planning failure. Figure 6.4(c) shows the precondition that failed: `free(jane)`



```
init
goto(agent,sword)
get(agent,sword)
goto(agent,dragon)
strike(agent,dragon,sword)
goto(agent,gold)
get(agent,gold)
goto(agent,axe)
get(agent,axe)
goto(agent,troll)
strike(agent,troll,axe)
goto(agent,bird)
get(agent,bird)
goto(agent,hag)
offer(agent,hag,and(gold,bird))
goto(agent,jane)
take_to_throne(agent,jane)
end
```

Figure 6.3: Experimental environment

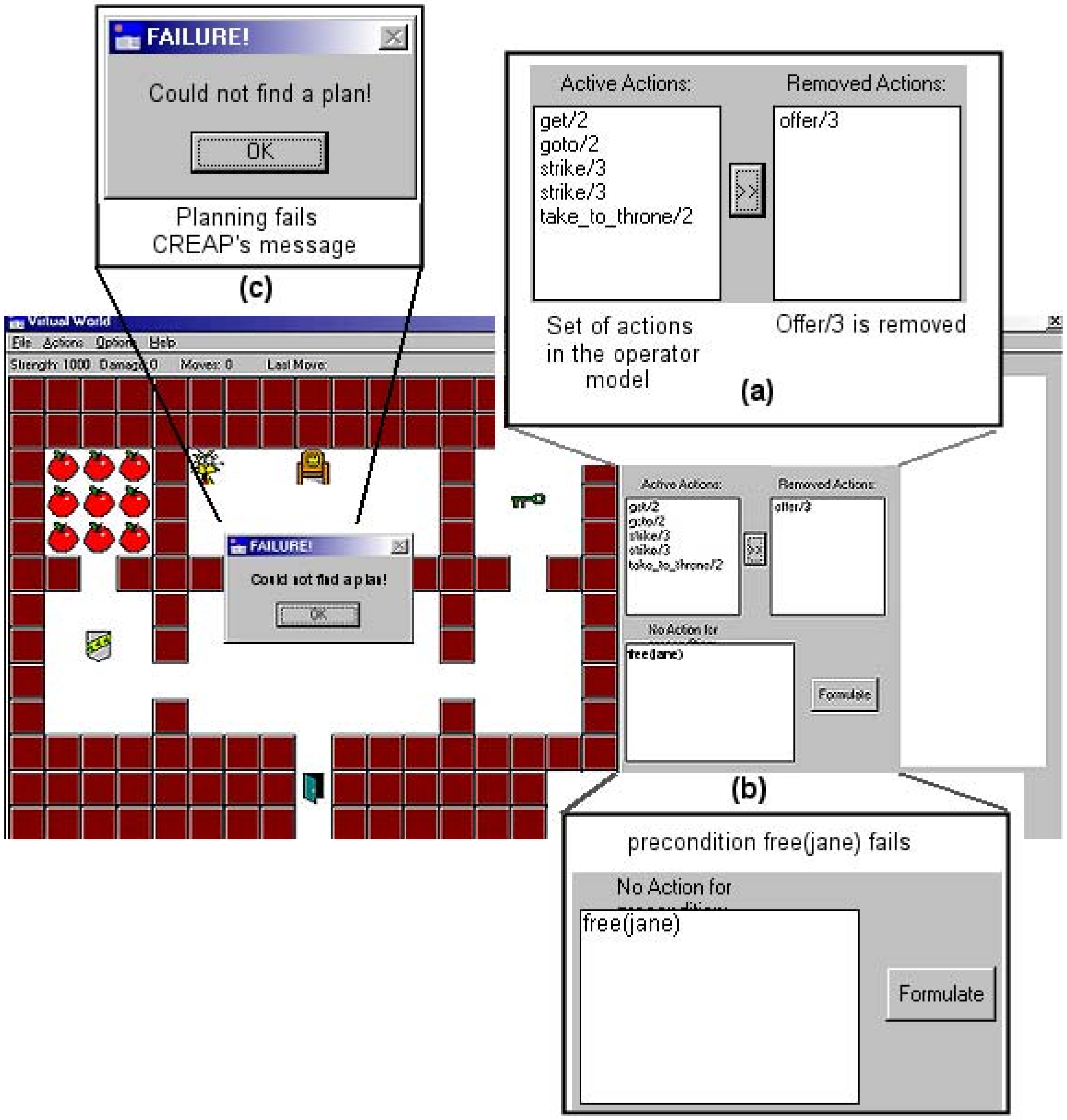


Figure 6.4: Experiment 2: When an action is removed from the operator model to make it incomplete and CREAP is demanded to plan

Now, we select the precondition that failed from the bottom of CREAP's interface and click on the "Formulate" button (see figure 6.5(a)). When the Formulate button is clicked, the improvisation module is invoked. The current-situation-generator makes a report of the failure and passes it to the analogical reasoner. The analogical reasoner retrieves a relevant case from the situation-repository and formulates a new action based on the structure-mapping, as we have described in Chapter 5.

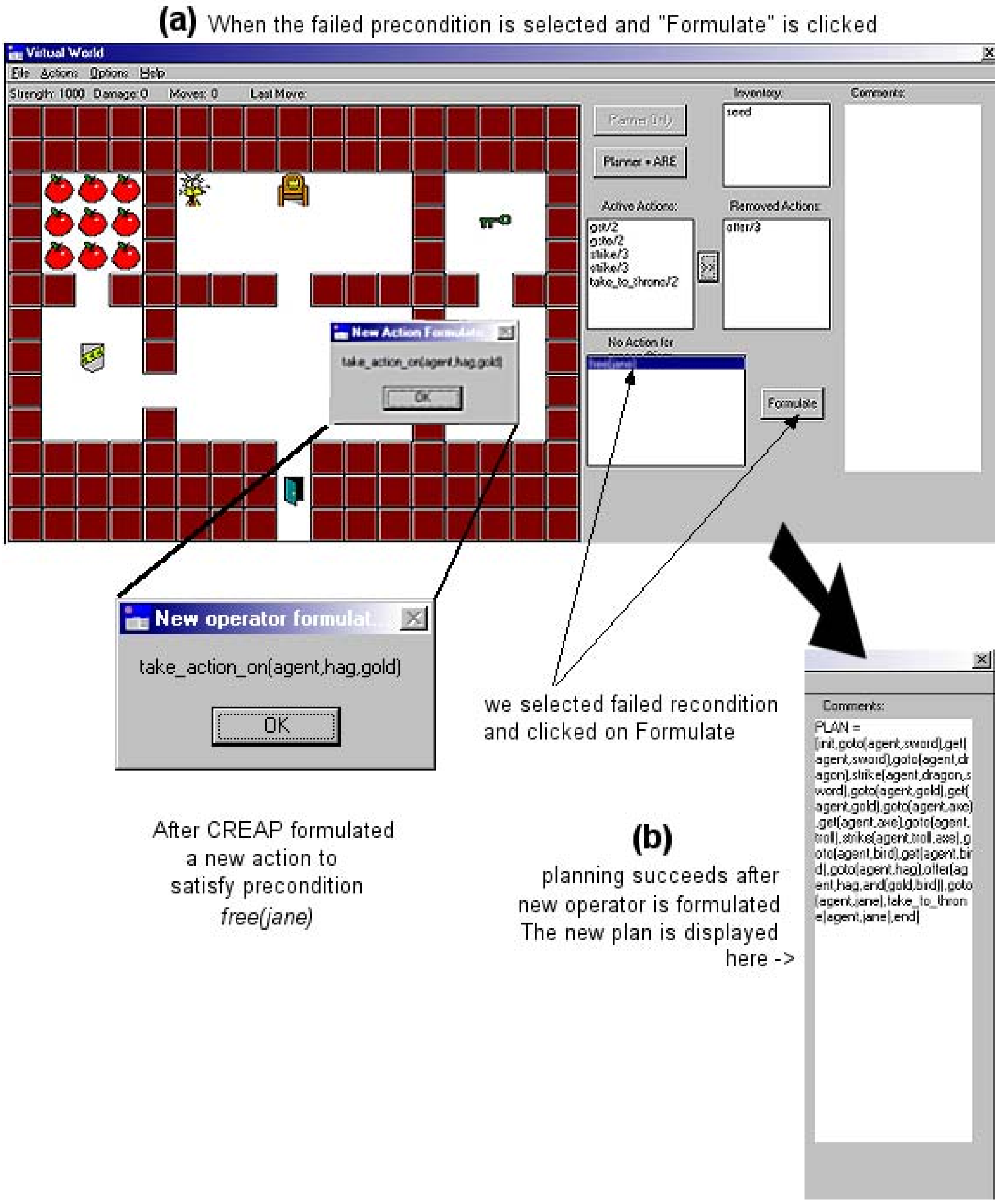
Figure 6.5(a) shows the result of clicking on the Formulate button (the result of improvisation). Note that a new action `take_action_on(agent,hag,gold)` was formulated and included as part of the operator model on a temporary basis.

If we ask CREAP to plan now, it succeeds. The new plan is displayed on the right-most side of CREAP's interface, as shown in figure 6.5(b). Apparently the newly formulated operator fills the gap left by the removal of the `Offer/3` operator in the beginning of the experiment. With the success of this improvisation process, CREAP has built a new high-level plan.

The following is the new plan generated by CREAP in this experiment:

```
[
  init, goto(agent,sword), get(agent,sword), goto(agent,dragon),
  strike(agent,dragon,sword), goto(agent,gold), get(agent,gold),
  goto(agent,hag), take_action_on(agent,hag,gold),
  goto(agent,jane), take_to_throne(agent,jane), end
]
```

Note the change in the new plan. The earlier plan was more elaborate. The new plan has fewer steps to perform. Now, is the new plan correct? That does not matter at this stage. At this stage we are only concerned with whether CREAP was able to produce a high-level plan or not, in spite of the incomplete operator model. Yes, it was able to. So, this is a significant improvement over classical planners. Further, the correctness of this plan can only be determined after CREAP executes it.


(a) When the failed precondition is selected and "Formulate" is clicked
Virtual World
File Actions Options Help
Strength: 1000 Damage: 0 Moves: 0 Last Move:
Planner Only
Planner + ARE
Inventory:
Comments:
Active Actions:
Removed Actions:
No Action for
Formulate
New Action Formulated
OK
New operator formulat...
we selected failed recondition and clicked on Formulate
After CREAP formulated a new action to satisfy precondition *free(jane)*
(b)
planning succeeds after new operator is formulated The new plan is displayed here ->
Comments:

take_action_on(agent,hag,gold)

Inventory: seed

Active Actions: get/2, goto/2, strike/3, strike/3, take_to_throne/2

Removed Actions: offer/3

No Action for: free(jane)

```
PLAN =
[init,goto(agent,sword),get(
agent,sword),goto(agent,dr
agon),strike(agent,dragon,s
word),goto(agent,gold),get(
agent,gold),goto(agent,axe)
,get(agent,axe),goto(agent,
troll),strike(agent,troll,axe),g
oto(agent,bird),get(agent,bir
d),goto(agent,hag),offer(ag
ent,hag,and(gold,bird)),goto
(agent,jane),take_to_thron
e(agent,jane),end]
```

Figure 6.5: Experiment 2: (a) When the failed precondition is selected and “Formulate” button is clicked. (b) Then when new operator is formulated, CREAP is asked to plan

Experiment 3

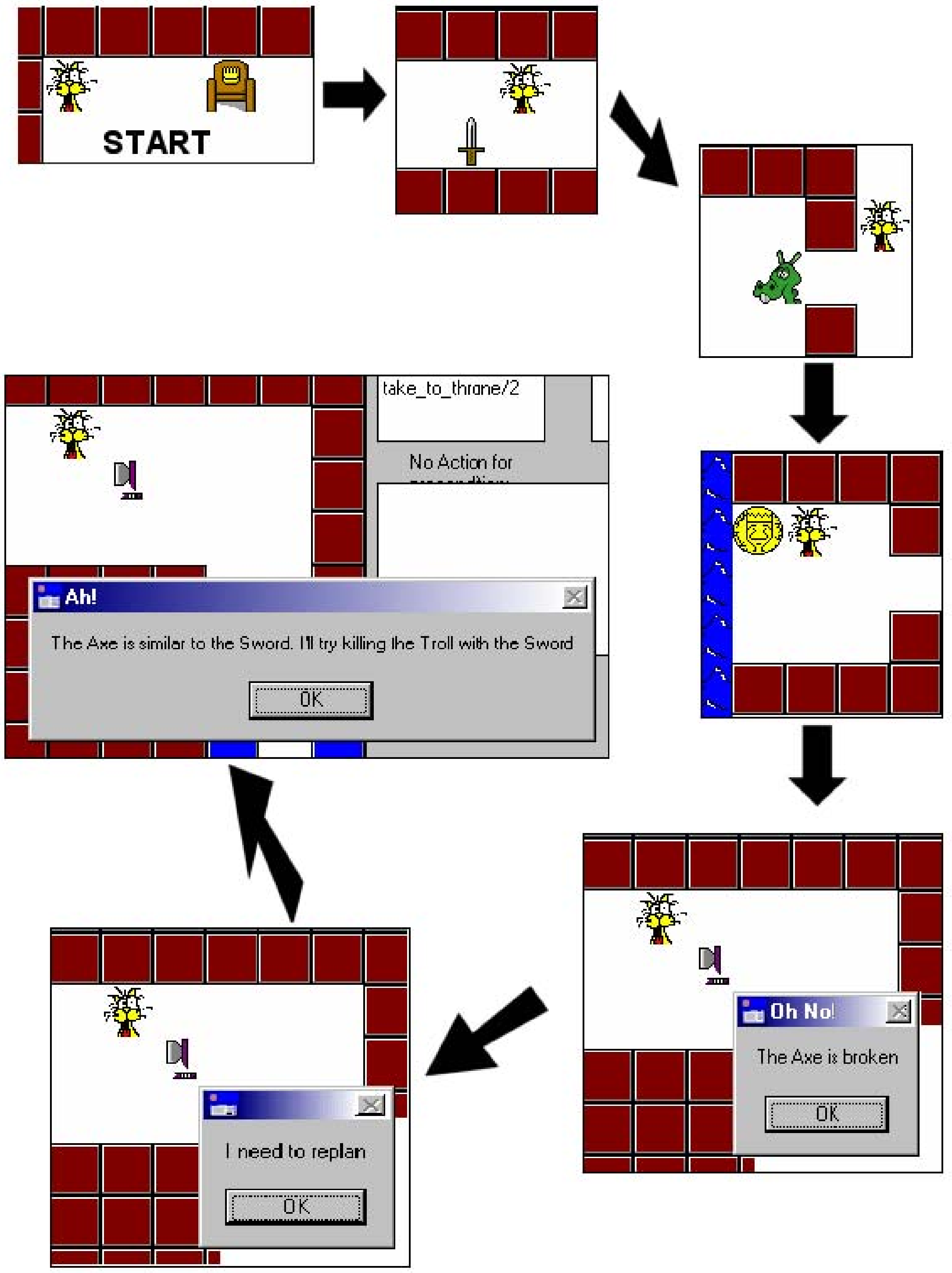
In experiment 2 we investigated improvisation from planning failure. In this experiment we are going to investigate improvisation from execution failure.

To reiterate, plan execution failures occur when the planning agent's domain model is incorrect. Its domain model may contain a fact f_1 which is represented as true, while in reality it is not so. Then the agent makes a plan based on this wrong fact. This plan is bound to fail at execution. For instance, I need a carry-bag for my new Laptop and my knowledge of superstores tells me that the Target superstore, located near my house, carries them. Based on this I make a plan to go to Target and purchase the bag. But when I get there I find out that they do not carry Laptop accessories or they are out of stock. So, my earlier plan fails during execution. After all, the real world is very dynamic and only partially accessible.

During the set-up for this experiment, we wanted to bring out such facets of the real world. Therefore we deliberately negated some fact in CREAP's domain model by changing its world. The change that we considered in this experiment was to make the axe unusable. Recollect that there is an axe in CREAP's world which it uses to kill the troll. We changed the world by replacing the good axe by a broken one. This negates the fact that CREAP can collect and use the axe.

In this experiment, CREAP makes an initial plan as usual. But the plan is made based on the knowledge that the axe is available. So, the initial plan consists of a step that involves grabbing the axe, before CREAP goes to slay the troll. Let us see how the execution of this plan goes, especially when we have replaced the unbroken axe in its world by a ruined one. Figure 6.6 takes us on a tour of CREAP's execution of the plan.

As we follow the execution we notice that when CREAP goes to get the axe it becomes aware of the broken axe. It communicates this to the observer through



Ah!

The Axe is similar to the Sword. I'll try killing the Troll with the Sword

Oh No!

The Axe is broken

I need to replan

Figure 6.6: Experiment 3: demonstration of precondition satisficing

the message-box “Oh No! The Axe is broken.” At this point the execution of the original plan fails because the axe, which was supposed to be intact, is no longer usable. CREAP then conveys its intent to improvise by saying “I need to replan.” The improvisation process kicks in, trying precondition satisficing initially.

The precondition which failed was `have(agent, axe)`. Recall that under such circumstances if the precondition has an object x, the agent looks for a similar object x' which can be substituted for x, based on the similarity between x and x'. The analogical reasoner in CREAP retrieves the sword as a close relative of the axe based on the following knowledge structures shown below:

```
% SWORD                            % AXE
?- defconcept(sword, [             ?- defconcept(axe, [
 [isa,0.5,weapon],                [isa,0.5,weapon],
 [action,0.5,thrust],             [action,0.5,chop],
 [attributes,0.5,long,sharp],     [attributes,0.5,sharp],
 [purpose,0.5,kill_victim]])      [purpose,0.5,chop_tree,kill_victim]])

% WEAPON
?- defconcept(weapon, [
   [isa, 0.5, instrument],
   [attributes, 0.5, dangerous],
   [purpose, 0.5, attack, self_defense, tool]]).

% KILL VICTIM                    % VICTIM
?- defconcept(kill_victim,       ?- defconcept(victim,
        [[isa,    0.5,  action],    [[isa, 0.5, person],
         [predicate,0.5,harm,kill], [attributes, 0.5, unfortunate]])
         [patient,  0.5,victim]])

% CHOP                             % THRUST
?- defept(chop,                    ?- defconcept(thrust,
   [[isa,0.5,mechanical_action],      [[isa, 0.5, mechanical_action],
   [result,0.5,wide_wound]])          [result, 0.5, deep_wound]])
```

One can clearly see the similarity between the axe and the sword looking at these knowledge structures. Therefore, CREAP improvises by substituting sword for axe and thereby trying to kill the troll using the sword. The result of the improvisation is conveyed by CREAP through the message “Ah! The axe is similar to the sword. I’ll try killing the Troll with the sword" (see figure 6.6). Thus, precondition satisficing was successful in adjusting the original plan to get a new improvised plan. However, this improvisation may or may not work. It can only be determined by executing this improvised plan. If CREAP is successful in actually killing the troll with the sword, it learns a new piece of information - that trolls are susceptible to swords too. Else, it might try to battle the troll with the sword and fail.

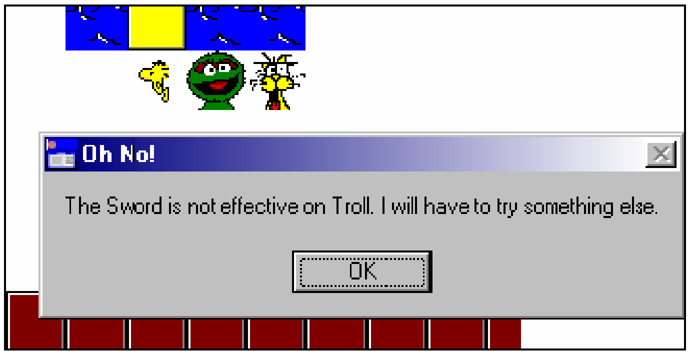
In any case, the objective of this experiment was to demonstrate the precondition satisficing process, which indeed went through.

Experiment 4

In experiment 3 we have demonstrated CREAP's ability to generate an improvised plan by precondition satisficing. However, we noted that the execution of this improvised plan need not always succeed. As a logical continuation of experiment 3, we will create conditions in the environment so that the improvised plan fails again. The objective of experiment 4 is to demonstrate the improvisation by *deep analogy* through new operator formulation, a behavior that can be considered a *creative act*.

In experiment 3, CREAP considered using sword as a weapon to kill the troll when it found that the axe was unusable. We will make this proposition false by programming the troll to be resistant to swords. As a result, CREAP will fail to kill the troll using the sword, forcing it to look for another solution leading to further improvisation. This time, however, CREAP can find no alternative way to perform precondition satisficing and resorts to new operator formulation using deep analogy.

Let us go back to following the execution after experiment 3. So, CREAP has decided to use the sword on the troll, and it already has the sword in its inventory. It then approaches the troll and finds that the troll cannot be killed by the sword. This is communicated to the observer through the message-box "Oh No! The Sword is not effective on the troll. I'll have to try something else" as depicted in figure 6.7. So, what just happened is the execution failure of the improvised plan from experiment 3. Now, CREAP has to improvise again.



The Sword is not effective on Troll. I will have to try something else.

Figure 6.7: Falure of execution of improvised plan of experiment 3

This time the goal for improvisation is `kill(agent, troll)`, because that is the condition which failed during execution. But let us not loose track of the higher goal that CREAP is trying to achieve. Why does CREAP want to kill the troll? To free the bird, guarded by the troll, so that CREAP can give the bird to the hag and free the princess. So, the goal is also `free(agent, bird)`.

As we have mentioned, the improvisation process begins by making a situation report of the failed situation. In this case the goals for the situation report are `kill(agent, troll)` and `free(agent, bird)`. The report should also contain the obvious fact that `guards(troll, bird)`.

A situation report is more involved than this. The function of the current-situation-generator is to take stock of the current situation, to look around in the world, to notice and gather more information about where the agent is and what it is doing. At this point, the current-situation-generator looks around and finds that CREAP is near a bridge. Notice the yellow blocks in figure 6.8, which is the bridge. Based on this it adds another fact `near(agent, bridge)` into the situation report.

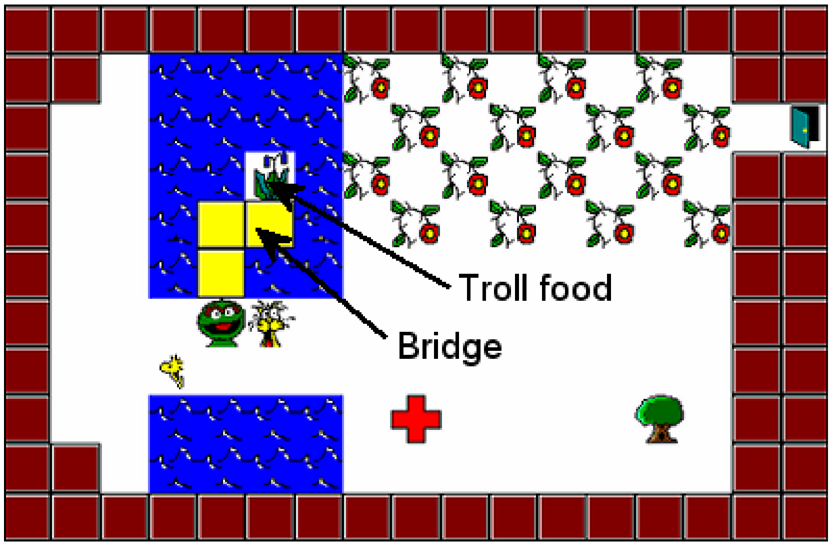


Figure 6.8: The bridge and Troll food

The current-situation-generator then queries the world (or nature) for more information about the bridge. Now, we have designed this world in such a way that the bridge leads to the troll-food (surrounded by water) and the bridge is the only means for the troll to get its food. If the bridge is destroyed, the troll is killed. In figure 6.8 notice the plant like thing at the end of the bridge (yellow blocks,) which is the troll food. So, when current-situation-generator queries the world (or nature) for more information about the bridge, it gets the following facts: `near(agent, bridge)`, `troll_depend_on(bridge,troll_food)`, `limited(troll_food)`.

Based on these facts and the goal that CREAP is trying to achieve, the current-situation-generator generates the following situation report:

```
[
   [goal, 0.5, free(bird)],
   [goal, 0.5, kill(agent,troll)],
   [fact, 0.5, near(agent,bridge)],
   [fact, 0.5, troll_depend_on(bridge, troll_food)]
```

```
    [fact, 0.5, limited(trollfood)],
]
```

This situation report is given to the analogical reasoner to find any other situation similar to this. This is like asking "What does my current situation remind me of from the past?" For this sort of improvisation, CREAP should be reminded of some situation in the past and use the knowledge from the past to seek a solution for the present. So, let us assume that CREAP had heard of a battle episode involving two parties X and Y.

In this episode, people of X were in the fort, which was completely surrounded by warriors of Y. The situation was more or less like what is depicted in figure 6.9. Party Y was stronger and was closing in on X. Therefore, X was in a very difficult situation and they had to find a way to stop Y. The fort was situated in a desert, so the food supply was limited. Y had carried their food supply with them and had stored it in a granary. So, some people from X sneaked into camp Y at night and destroyed the granary. This led to Y running out of its food supply and having to retreat.

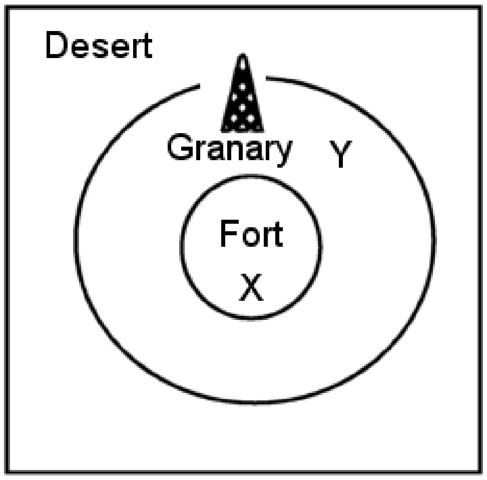


Figure 6.9: The battlefield scenario

This episode is stored in CREAP's memory (situation repository) as shown in figure 6.10. Well, this is not the actual symbolic representation, but a graphical representation of the contents of the situation. The actual knowledge structure does capture the causal structure and also the facts menioned in the figure.

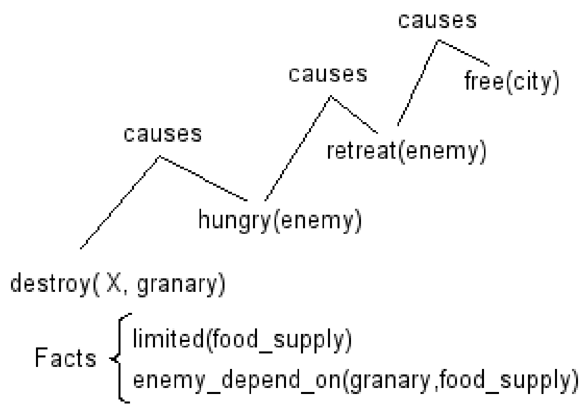


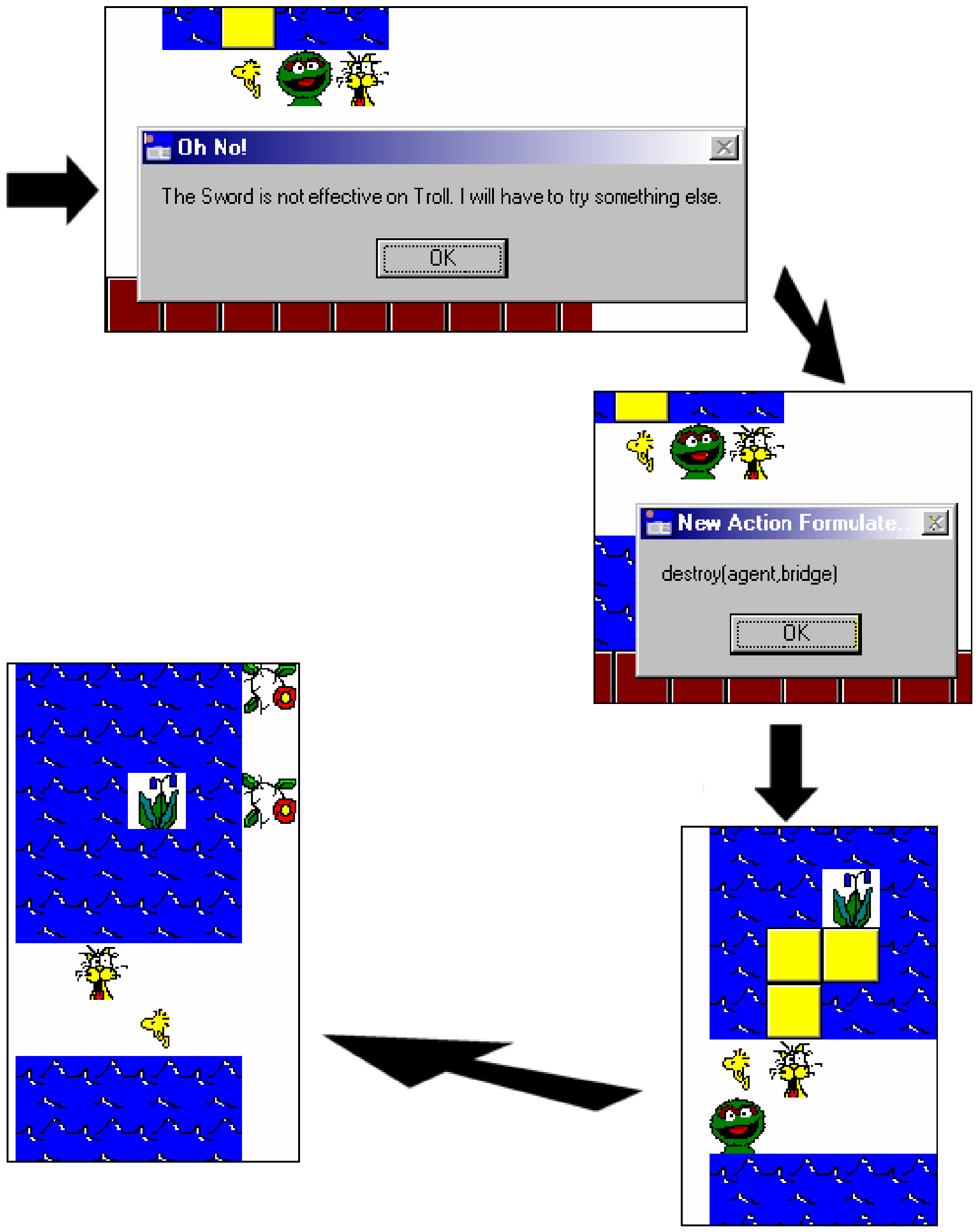
Figure 6.10: Representation of battlefield scenario

When the situation report is given to the analogical reasoner, the analogical reasoner retrieves the above episode with the following mappings based on the structural similarity:

```
Agent <-> X
Troll <-> Enemy
Granary <-> Bridge
Bird <-> City
Troll_food <-> Food_supply
```

Based on this the analogical reasoner suggests a new action for CREAP `destroy(agent, bridge)` because the action `destroy(granry)` had worked for X in the past, and CREAP is in the same situation as X was.

Figure 6.11 shows the execution from after the new operator was formulated. Notice that CREAP has communicated its intension to destroy the bridge to the observer through the message-box "New Action Formulated! destroy(agent, bridge)."



The Sword is not effective on Troll. I will have to try something else.

destroy(agent,bridge)

Figure 6.11: Representation of battlefield scenario

In the next step it has to go to the base of the bridge to destroy it. In the next snapshot (c), one can notice that CREAP is in a cell directly below the yellow block, which shows that it is in a position from where it can destroy the bridge. In the next snapshot (d), the yellow blocks have disappeared and the troll is not visible. This shows that CREAP destroyed the bridge and the troll is killed. Therefore, the improvisation worked in this case.

Experiment 4 shows how CREAP formulated a new operator `destroy(agent, bridge)` which was not a part of the operator model before. This operator worked in this situation and CREAP was successful in killing the troll, and freeing and collecting the bird. The rest of the plan execution (bribing the Hag and rescuing the princess) is the same as in experiment 1 - figure 6.3.

6.5 Summary

This Chapter has presented four experiments which are examples of CREAP engaged in improvised behavior. These experiments highlight the architecture of CREAP, demonstrating its ability to reproduce and deal with characteristics of improvisational behavior in planning. The agent is shown relying on background knowledge dealing with overlapping and competing constraints; taking advantage of fortuitous circumstances in the environment around itself; and using perception to its advantage. Moreover, it illustrates that a goal-directed representation is ideally suited to the dynamic nature of improvisation, and demonstrates the power of these goals in practice.

Having said this, there are still many inadequacies to this implementation. Most significantly, it uses only primitive knowledge representation for situations (shown in figure 6.10), which is rather rudimentary, and not as detailed as what might be the case in humans. Also, these situations are represented as independent isolated entities in the memory and are not connected to form a more comprehensive memory. Also this memory is static, i.e., these representations in the memory are not re-organized as a function of new experiences.

Despite all these inadequacies, however, this implementation demonstrates the range of complex behavior CREAP is capable of exhibiting, and will serve as a useful prototype for more extensive examples of improvised behavior as well as for experimenting with many of the processes underlying improvisation, including goal-directed real-time control and reasoning about function and purpose.

Chapter 7

Conclusion and future work

This thesis began with the argument that existing theories of planning, while being well-suited to specialized and restricted domains, were inadequate for dealing with incompleteness and incorrectness problems. An analysis of how humans plan and perform in the real world illustrates that they cope with incompleteness and incorrectness by constantly improvising in the face of failure using their background knowledge. Based on this, we came-up with an architecture for planning systems which can incorporate an improvisational process.

We talk about the improvisation process as suggesting new solutions to failed planning situations, not based on deliberative first-principles reasoning but on unanticipated analogical reasoning. By basing the improvisational process on analogical reasoning, we can explain how people come up with mundane or novel reactions to planning failures.

We have designed a softbot called CREAP, which incorporates our architecture, and tested it by observing its behavior as it performs in a simulation testbed. Thus, this research lies at an interesting crossroad of three important research areas in artificial intelligence: AI planning, analogical reasoning and testbed systems. Chapter 2 provides a detailed background of these three areas of research and compares and contrasts the various approaches that are relevant to our approach. In Chapter 2 we also discuss the important shortcomings of some of the other approaches and how we are trying to address those shortcomings using our architecture.

In Chapter 3 we give an overview of CREAP, our agent, and its environment of operation. We also discussed three key points that we will have to consider while designing an environment for CREAP so that it can model certain facets of the real world, in order to provide sufficient opportunities for CREAP to exhibit improvisational behavior; the key-points being: (1) complexity of information, (2) inclusion of dynamic elements in the environment, and (3) domain extensibility.

In Chapter 4 we looked at the various components that make up CREAP's architecture and discussed the functionality of each of them. These components are: the STRIPS-based planner, the domain and operator models, the current-situation-generator, the analogical reasoner and the situation repository. Although we provided a fairly detailed picture of each of these components, we decided to dedicate another chapter to discuss how these components came together; we were skeptical that the big picture was lost owing to the reductionist view of this chapter.

In Chapter 5 we discussed our architecture from a process point of view, which showed where each component contributed to the entire process. This sort of provided the big picture. We discussed the planning process and the improvisational process and how the two processes interacted with each other leading to the more complex overall behavior of CREAP. We also talked about two modes of improvisational process: precondition satisficing and new operator formulation.

In Chapter 6 we looked at experimentation and observation. Here we discussed the experimental set-up and the four experiments which show the complex behavior of CREAP. These experiments showed the range of complex behavior CREAP is capable of exhibiting.

Future work

As Chapter 1 has described, intelligent agency is a field with connections to virtually all of artificial intelligence. Because of these connections, there is a great deal of future research suggested by the CREAP architecture, both including and beyond its limitations.

Within the limitations previously described, the most significant to be addressed is the improvement of the implemented representations for situations in CREAP's memory or situation repository. As of now, we have hand-coded these situations as independent, isolates situations, much like the records in a database. However, some of the current models of human memory suggest that it is dynamic and constantly reorganizing as a function of time and new experiences. This is an important point to consider: how does memory evolve?

Another important area of work that emerges as a result of this work is monitoring and explaining failures. When a planning failure occurs, we never bother to explain the failure in terms of existing knowledge in this architecture. If one tries to explain a failure then some facts become clear which were not considered earlier. Or this may lead to reexamining the assumptions and constructive reflection. To put it simply, explaining failure might itself lead to solutions. This reflective component is missing in the architecture.

The next limitation is that the interface between the planning and improvisation components is not clean. As of now, it is more serial in fashion, more or less like an assembly-line. The planner fails. The current-situation-generator takes over and does its bit. Then the control goes to the analogical reasoner, and so on. This is not such a clean model.

Further, as an extension to the existing architecture, we intend to work on marrying the improvisational process with other types of planners. Right now, we have restricted ourselves to the classical planners. But we mentioned in Chapter 2 that there are other fundamentally different planners like memory-based planners and reactive planners. We intend to investigate the possibility of extending the existing architecture to work with different types of planners.

Another area of improvement that needs to be considered is the perception and action models for CREAP. Right now, the perception and action models are quite rudimentary. Also, we assume that CREAP has a map of its entire world. This assumption needs to be reconsidered and in future experiments CREAP should explore the world on its own and should have the capability to make a map on its own.

Finally, with regard to experimentation, we have only presented 4 experiments and conducted a few more. This is insufficient to comprehensively demonstrate the goodness of our approach. We have to take a more statistical approach and compare our approach to others through a lot more experiments and establish the goodness of our approach. Also, we have conducted the experiments in only one world. We intend to design and build a lot more worlds, which are more complex and study the performance of CREAP in all of these.

Appendix A

A sample representation of a scenario

The following shows the actual representation of one of the scenarios (dragon-gold scenario) stored in the situation repository.

```
?- defconcept(dgs,
   [[goal,  0.5, dgs_free_gold],
    [event, 0.5, dgs_goaway_dragon],
    [event, 0.5, dgs_take_action_agent_dragon_sword],
    [fact,  0.5, dgs_near_agent_dragon],
    [fact,  0.5, dgs_has_agent_sword],
    [fact,  0.5, dgs_goes_away_with_dragon_sword],
    [fact,  0.5, dgs_guards_dragon_sword]]).

?- defconcept(dgs_cause1,
   [[predicate, 0.5, cause],
    [patient,   0.5, dgs_free_gold],
    [agent,     0.5, dgs_goaway_dragon]]).

?- defconcept(dgs_cause2,
   [[predicate, 0.5, cause],
    [patient,   0.5, dgs_goaway_dragon],
    [agent,     0.5, dgs_take_action_agent_dragon_sword]]).

?- defconcept(dgs_free_gold,
   [[predicate, 0.5, free],
    [patient,   0.5, dgs_gold],
    [agent,     0.5, dgs_agt]]).

?- defconcept(dgs_goaway_dragon,
   [[predicate, 0.5, go_away],
    [arg,       0.5, dgs_dragon]]).

?- defconcept(dgs_take_action_agent_dragon_sword,
   [[predicate, 0.5, take_action],
    [patient,   0.5, dgs_dragon],
    [agent,     0.5, dgs_agt],
    [obj,       0.5, dgs_sword]]).

?- defconcept(dgs_near_agent_dragon,
   [[predicate, 0.5, near],
    [patient,   0.5, dgs_dragon],
    [agent,     0.5, dgs_agt]]).

?- defconcept(dgs_has_agent_sword,
   [[predicate, 0.5, has],
    [patient,   0.5, dgs_sword],
    [agent,     0.5, dgs_agt]]).

?- defconcept(dgs_goes_away_with_dragon_sword,
   [[predicate, 0.5, goes_away_with],
    [patient,   0.5, dgs_sword],
    [agent,     0.5, dgs_dragon]]).

?- defconcept(dgs_guards_dragon_sword,
   [[predicate, 0.5, guards],
    [patient,   0.5, dgs_gold],
    [agent,     0.5, dgs_dragon]]).

?- defconcept(dgs_agt,
   [[isa, 0.5, leaf]]).

?- defconcept(dgs_gold,
   [[isa, 0.5, leaf]]).

?- defconcept(dgs_sword,
   [[isa, 0.5, leaf]]).

?- defconcept(dgs_dragon,
   [[isa, 0.5, leaf]]).
```

Bibliography

[1] Allen, James; Hendler, James; and Tate, Austin. (1990) *Readings in Planning.* San Mateo, CA: Morgan Kaufmann.

[2] Anderson, John E. (1995) *Constraint-Directed Improvisation for Everyday Activities* PhD thesis, Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, Canada.

[3] Anderson, John R. (1995) *Cognitive Psychology and its implications.* New York: Worth Publishers.

[4] Bridger, B. C.; Crouch, David; and Donald Nute. (2001) Planning Agent Architecture for a Virtual World Environment. Proceedings, IC-AI-2001, Monte Carlo Resort, Las Vegas, Nevada, 3:1059-1065. CSREA Press.

[5] Carbonell, Jaime G.; and Yolanda Gil. (1990) Learning by experimentation: The operator refinement method. In Michalski, R. S., and Kodratoff, Y., eds., *Machine Learning: An Artificial Intelligence Approach* 3:191–213. Palo Alto, CA: Morgan Kaufmann.

[6] Chapman, David. (1987) Planning for Conjunctive Goals. *Artificial Intelligence* 32: 333-377.

[7] Cheng, Patricia; and Carbonell, Jaime G. (1986) The FERMI System: Inducing Iterative Macro-Operators from Experience. Proceedings, AAAI-86: 490-495.

[8] Engelson, Sean P. and Bertani, Niklas. (1992) *Ars Magna: The Abstract Robot SimulatorManual*, version 1.0. Department of Computer Science technical report, Yale University, New Haven, Connecticut.

[9] Falkenhainer, Brian; Forbus, Kenneth D.; and Gentner, Dedre. (1989) The structure mapping engine: Algorithm and example. *Artificial Intelligence* 41:1-63.

[10] Forbus, Kenneth D.; Gentner, Dedre; and Law, Keith. (1995) MAC/FAC: A Model of Similarity-Based Retrieval. *Cognitive Science* 19:141-205.

[11] Gentner, Dedre. (1982) Are scientific analogies metaphors? In Miall, David S., eds., *Metaphor: Problems and perspectives* 106-132. Brighton: Harvester Press.

[12] Gentner, Dedre. (1983) Structure-mapping: A theoretical framework for analogy. *Cognitive Science* 7:155-170.

[13] Gick, M. L.; and Holyoak, K. J. (1983) Analogical problem solving. *Cognitive Psychology* 12:306-335.

[14] Gil, Yolanda. (1992) *Acquiring Domain Knowledge for Planning by Experimentation.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

[15] Guha R. V; and Lenat D. B. (1990) *Cyc: A Midterm Report.* AI Magazine, 11:32–59.

[16] Hall, Rogers. P. (1989) Computational approaches to analogical reasoning. *Artificial Intelligence* 39:39-120

[17] Hofstadter, Douglas. (1995) *Fluid concepts and creative analogies.* New York: Basic Books.

[18] Hofstadter, Douglas. (1984) *The copycat project: An experiment in nondeterminism and creative analogies.* A.I. Memo 755, Artificial Intelligence Laboratory, MIT, Cambridge, MA.

[19] Holyoak, K., Gentner, D., Kokinov, B. (2001) The Place of Analogy in Cognition. In Gentner, D., Holyoak, K., Kokinov, B., eds. *The Analogical Mind: Perspectives from Cognitive Science.* Cambridge, MA: MIT Press.

[20] Holyoak, Keith J.; and Thagard, Paul. (1989) Analogical Mapping by Constraint Satisfaction. *Cognitive Science*, 13.3:295–355.

[21] Iwamoto, Masahiko. (1994) A planner with quality goal and its speedup learning for optimization problem. Proceedings, Second International Conference on AI Planning Systems, Chicago, IL, 281-286.

[22] Kedar-Cabelli, Smadar (1988) Analogy - from a unified perspective. In Helman D. H., eds.,*Analogical Reasoning* 65–103. Kluwer Academic Publishers.

[23] Kolodner, Janet. (1993) *Case-based reasoning.* San Mateo, CA: Morgan Kaufmann.

[24] Korf, Richard E. (1985) Macro-Operators: A weak method for learning. *Artifcial Intelligence* 26.1:35-77.

[25] Minton, Steven. (1988) *Learning Effective Search Control Knowledge: An Explanation-Based Approach.* Boston: Kluwer Academic Publishers.

[26] Mitchell, M. (1993) *Analogy-making as perception.* Cambridge: MIT Press.

[27] Pollack, Martha E.; and Ringuette, M. (1990) Introducing the Tileworld: Experimentally evaluating agent architectures Proceedings, AAAI-90, 183–189.

[28] Ross, B. H. (1984) Remindings and their effects in learning a cognitive skill. *Cognitive Psychology*, 16:371-416.

[29] Ruby, David; and Kibler, Dennis. (1990) Learning steppingstones for problem solving. Proceedings, Darpa Workshop on Innovative Approaches to Planning, Scheduling and Control, San Diego, CA, 366-373.

[30] Saunders, R. *Hybrid Evolutionary Algorithms in Creative Design*. [On-line]. Available: http://www.arch.usyd.edu.au/~rob/study/Hybrid.html.

[31] Schank, Roger C. (1982) *Dynamic Memory*. New York: Cambridge University Press.

[32] Simon, Herbert A. (1982) Rational Choice and the Structure of the Environment Simon, In Herbert A. eds. *Models of Bounded Rationality* 2:259-268. Cambridge, MA: MIT Press

[33] Stuart, Russell; and Norvig, Peter. (1995) *Artificial Intelligence: A Modern Approach*. New Jersey: Prentce Hall.

[34] Wang, Xuemei. (1995) Learning by observation and practice: An incremental approach for planning operator acquisition. Proceedings, Twelfth International Conference on Machine Learning, Tahoe City, CA, 549–557.

[35] Wilensky, Robert. (1983) *Planning and Understanding*. Reading, MA: Addison-Wesley Publishing Company.

[36] Wilkins, David E. (1988) *Practical Planning: Extending the Classical AI Planning Paradigm*. San Mateo, CA: Morgan Kaufmann Publishers, Inc.

[37] Winston, P. H. (1980) Learning and reasoning by Analogy. *Communications of the ACM* 23:689-703.

[38] Veale, T. (1995) *Metaphor, Memory and Meaning: Symbolic and Connectionist Issues in Metaphor Comprehension.* PhD thesis, Trinity College, Dublin

[39] Veale, T. (1997) *A Metaphor Review by the Author.* [On-line]. Available: http://www.compapp.dcu.ie/~tonyv/trinity/author.html.

[40] Veleso, Manuela M. (1994) *Planning and Learning by Analogical Reasoning.* Berlin: Springer Verlag.