

**AUTOMATING REQUIREMENTS ENGINEERING USING  
ARTIFICIAL INTELLIGENCE PLANNING TECHNIQUES**


by

**JOHN S. ANDERSON**

**A DISSERTATION**

**Presented to the Department of Computer  
and Information Science  
and the Graduate School of the University of Oregon  
in partial fulfillment of the requirements  
for the degree of  
Doctor of Philosophy**

**March 1993**

APPROVED:   
Stephen F. Fickas

## An Abstract of the Dissertation of

John S. Anderson for the degree of Doctor of Philosophy  
in the Department of Computer and Information Science to be taken March 1993

Title: AUTOMATING REQUIREMENTS ENGINEERING USING ARTIFICIAL  
INTELLIGENCE PLANNING TECHNIQUES

Approved:

  
Dr. Stephen F. Fickas

Requirements engineering is a critical and yet poorly understood aspect of software engineering and other complex design tasks. In this dissertation I focus on requirements engineering for a particular class of designed artifacts, referred to as reactive systems, which provide services in response to events in the environment. For this class of systems, the requirements engineering task is to identify the set of services that best satisfy the client's requirements, and to ensure that the client understands and agrees to any trade-offs between competing concerns such as functionality, performance, ease of use, and cost. Coming to consensus requires bridging the gap between the client's requirements, which describe states in the application domain, and the functional specification, which describes the services to be provided by the target artifact.

My thesis is that the process of constructing and validating a functional specification of a reactive system can be carried out using artificial intelligence planning techniques. The client's requirements can be expressed as a set of planning problems: finding a sequence of actions that lead from an initial state to a goal state. The sequence of actions required to reach the goal state may include actions to be performed by the target artifact. These actions determine the services that must be provided by the target artifact.

I have implemented an automated planner and used it to explore the role of planning in the requirements engineering process. In this dissertation I first describe how requirements engineering can be viewed as a planning problem in which actions are composed into sequences that achieve goal states. I then show how the automated planner can be used in several parts of the requirements engineering process: proposing action sequences that satisfy the requirements, constructing functional specifications based on those sequences, and critiquing the functional specifications by showing violations of client restrictions. I describe an extended example of using the automated planner to develop a functional specification of a benchmark problem, a library database. I evaluate the strengths and weaknesses of the planning approach to requirements engineering based on experience with the automated planner.

**VITA**

**NAME OF AUTHOR:** John Searles Anderson

**PLACE OF BIRTH:** Missoula, Montana

**DATE OF BIRTH:** June 13, 1957

**GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:**

University of Oregon  
University of Montana  
Montana State University  
Stanford University

**DEGREES AWARDED:**

Doctor of Philosophy, 1993, University of Oregon  
Masters of Science, 1986, University of Oregon  
Bachelor of Science, 1981, Montana State University

**AREAS OF SPECIAL INTEREST:**

Application of Artificial Intelligence to Requirements Engineering and Design

**PROFESSIONAL EXPERIENCE:**

Teaching Assistant, Department of Computer and Information Science, University of Oregon, Eugene, 1990-93

Research Assistant, Department of Computer and Information Science, University of Oregon, Eugene, 1987-90

**AWARDS AND HONORS:**

National Science Foundation Graduate Fellowship, 1984-87

## PUBLICATIONS:

- Anderson, J.S. (1990). Partial commitment and requirements engineering. Working notes of the AAAI Workshop on Planning in Complex Domains, Boston.
- Anderson, J. S. (1991). The composition model of specification: a comparative study. CIS-TR-91-03. University of Oregon.
- Anderson, J.S. and Durney, B. (1991). Using scenarios for composing specifications. Workshop on Model-based Reasoning, AAAI-91, Anaheim.
- Anderson, John S. and Durney, B. (in press). Critiquing in systems design and specification. To appear in *The Knowledge Engineering Review*, special issue on expert critiquing systems.
- Anderson, J.S. and Farley, A.M. (1988). Plan abstraction based on operator generalization. In Proceedings of the 1988 AAAI Conference, St. Paul, 100-104.
- Anderson, J.S. and Farley, A.M. (1990). Incremental selection in plan composition. CIS-TR-90-11, University of Oregon.
- Anderson, J. S. and Fickas, S. (1989). A proposed perspective shift: viewing specification design as a planning problem. In Proceedings of the 5th International Workshop on Software Specification and Design. Also appears in Partridge (ed.), Artificial Intelligence and Software Engineering, Ablex, 1991.
- Anderson, J.S. and Fickas, S. (1990). Using approximations and abstraction in software engineering, in Proceedings of the AAAI Workshop on Automatic Generation of Approximations and Abstractions, AAAI-90, Boston.
- Fickas, S., Anderson, J., Robinson, W. (1990, December). The Kate project: supporting specification construction, Technical Report 90-24, Computer Science Department, University of Oregon.

## ACKNOWLEDGMENTS

I wish to express my thanks to Steve Fickas for providing a rich and active research environment. I thank Art Farley for many stimulating discussions. I thank Mike Posner for his advice and encouragement at the beginning and the end of my graduate career. I also thank the members of the requirements engineering research group: Bill Robinson, Brian Durney, Rob Helm, and Anne Dardenne Helm.

My research was supported by the National Science Foundation through a Graduate Research Fellowship and through grant #CCR-8804085.

## TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION . . . . .	1
Specifying the Services to be Provided by a Functional Artifact . . . .	2
The Planning Approach . . . . .	11
Motivation for This Research . . . . .	15
Research Goals . . . . .	18
Overview of the Dissertation . . . . .	20
II. FORMULATING THE SPECIFICATION ENGINEERING PROBLEM	24
Planning: Overview . . . . .	25
Specification Engineering . . . . .	33
Client Requirements . . . . .	36
Output: Functional Specification . . . . .	42
Domain Model . . . . .	44
Summary: Defining the Specification Engineering Task . . . . .	46
III. THE PLANNING APPROACH TO SPECIFICATION ENGINEERING	50
Decomposing a Specification Engineering Problem into Planning Tasks	51
Composition . . . . .	58
Detect Safety Violations . . . . .	63
Address Safety Violations . . . . .	66
Summary: The Planning Approach to Specification Engineering . . .	71
IV. AVOID SEARCH USING STRUCTURED KNOWLEDGE . . . . .	73
Knowledge Organization . . . . .	74
Incremental Selection . . . . .	77
Abstract Plans . . . . .	87
Judgment: Evaluate Options Using Stored Selection Information . . .	90
Extensions to Specification Engineering . . . . .	94
Summary . . . . .	97
V. RESULTS AND EVALUATION . . . . .	98
Results . . . . .	100
Representation . . . . .	110
Clarification: Resolving Ambiguity . . . . .	112



	Page
Resolving Incompleteness . . . . .	117
Correctness: What Can We Guarantee?. . . . .	121
Performance and Scalability . . . . .	123
Summary: Evaluation of Results . . . . .	127
<b>VI. CONCLUSION . . . . .</b>	<b>129</b>
Research Summary . . . . .	129
Benefits of Research . . . . .	137
Issues and Limitations . . . . .	139
Future Work. . . . .	140
Conclusions . . . . .	141
<b>APPENDIX . . . . .</b>	<b>142</b>
<b>BIBLIOGRAPHY . . . . .</b>	<b>154</b>

## LIST OF FIGURES

Figure		Page
1.1	The Development Process . . . . .	5
1.2	Inputs and Outputs of the Specification Engineering Process. . . . .	8
2.1	The Purpose of Planning Is to Find a Sequence of Actions. . . . .	26
2.2	Example Operator: Check Out . . . . .	28
2.3	Inputs and Outputs of the Planning Process. . . . .	30
2.4	Each Node in the Search Space Represents a Different Sequence of Actions	31
2.5	Inputs and Outputs of the Specification Engineering Process. . . . .	34
2.6	Each Node in the Search Space is a Functional Specification . . . . .	35
2.7	Requirements Can be Expressed as State Transitions. . . . .	37
2.8	Initial Statement of Requirements for the Library Problem. . . . .	38
2.9	An Achievement Requirement . . . . .	41
2.10	A State to be Maintained . . . . .	41
2.11	Requirements for the Library Problem . . . . .	43
2.12	Operators Which Make Up the Functional Specification of the Artifact .	44
2.13	Example Operators Representing Actions of Actors in the Environment.	47
2.14	Overview of the Specification Engineering Process: Inputs and Outputs.	48
3.1	Algorithm for Solving Problems Using Deficiency Driven Techniques .	52
3.2	Deficiency Driven Specification Engineering. . . . .	53
3.3	A Desired State Transition Which Cannot Be Achieved . . . . .	54
3.4	A Prohibited State Transition Which Can Be Achieved . . . . .	55
3.5	The Relationship Between Requirements and Scenarios . . . . .	56
3.6	Composition Phase . . . . .	59
3.7	Example Achievement Requirement from the Library Problem . . . . .	60
3.8	Example Plan for Checking Out a Book . . . . .	61
3.9	Operators that Represent Services Are Placed into the Specification. . .	62
3.10	Representative Plans that Might be Produced. . . . .	63
3.11	The Critiquing Phase . . . . .	64

	Page
3.12 Example Safety Requirement for the Library Problem . . . . .	65
3.13 Inserting a Guard to Prevent a Prohibited Transition . . . . .	68
4.1 Generalization Hierarchy of Operators . . . . .	75
4.2 Schemas are Decomposed into More Detailed Components . . . . .	76
4.3 Detecting Interactions Efficiently: Incremental Selection . . . . .	82
4.4 An Empty Slot in a Plan Element is Filled . . . . .	84
4.5 A Plan Element is Specialized. . . . .	85
4.6 Plan Composition Algorithm . . . . .	90
5.1 Original Problem Statement for the Library Problem. . . . .	100
5.2 Overview of the Manual Library System . . . . .	101
5.3 Overview of the Automated Library System . . . . .	101
5.4 Problem Definition for Completing a Project Using a Library Book. . .	102
5.5 Plan for Completing a Project - Manual Record-keeping System . . . .	103
5.6 Plan for Completing a Project - Automated Record-keeping System. . .	104
5.7 Problem: Two Patrons have Projects which Require the Same Book. . .	104
5.8 Plan for Two Patrons to Complete Projects Requiring the Same Book. .	105
5.9 Problem Definition for Finding out that a Book is not Available. . . . .	106
5.10 Plan for Finding out that a Book is not Available in the Manual Library.	106
5.11 Plan for Finding out that a Book is not Available in the Automated Library	107
5.12 Plan Showing Violation of Borrowing Limit Requirement . . . . .	108
5.13 Revised Operator Representing a Restricted Form of Check Out. . . . .	108
5.14 Using the Restricted Check-out Operator, the Planner Fails . . . . .	109
5.15 Representation of Tasks as Initial and Final States . . . . .	111
5.16 Three Different Interpretations of the Requirement. . . . .	115

## CHAPTER I

### INTRODUCTION

The work reported here addresses the formalization and automation of portions of the requirements engineering process. In particular, this dissertation focuses on the construction and validation of a functional specification of a target artifact based on the objectives and preferences expressed by a client.

My thesis is that artificial intelligence planning techniques can be used to automate portions of the requirements engineering process. One consequence of this thesis is that much of the existing work in the domain of automated planning can be applied to requirements engineering problems. As a result, we can avoid a great deal of duplicated effort. A primary goal of this dissertation is to show how automated planning techniques can be applied to requirements engineering. However, requirements engineering introduces its own set of issues. A secondary goal of this dissertation is to identify the areas in requirements engineering where standard planning methods fall short and new techniques are needed.

I argue that with the right perspective and set of assumptions, work in AI planning may be brought to bear on three aspects of requirements engineering: proposing a functional specification, analyzing a proposed specification for deficiencies, and modifying a specification to remove deficiencies. I describe a program called OPIE which automates portions of the requirements engineering process.

In the following section I begin by describing the scope of this work, reviewing the requirements engineering process and defining the terms I will use throughout the dissertation. In the next section I introduce the planning approach that I have taken. I then dis-

Discuss the benefits of formalizing and automating requirements engineering, describe the contributions of this research and present an overview of the rest of the dissertation.

### Specifying the Services to be Provided by a Functional Artifact

Requirements engineering is one of the early phases in the development of functional artifacts. In this dissertation I focus primarily on a particular sub-process of requirements engineering, which I refer to as specification engineering.

In this dissertation, I define specification engineering as the process of composing and validating a functional specification. A functional specification is an abstract model of an artifact being developed. The functional specification describes the services to be provided by the artifact. A service is an externally observable action of the artifact in response to a stimulus from its environment. For example, an elevator provides services such as moving from floor to floor and opening and closing its doors. A primary goal of specification engineering is to ensure that the artifact provides the services the client needs.

#### Scope: What Kinds of Problems Does the Planning Approach Address?

The approach described here applies to a particular class of requirements engineering problems, those concerned with reactive systems. Reactive systems are artifacts that provide a variety of services in response to events in the environment. Reactive systems contrast with transformational systems, in which all inputs are supplied at once and no further interaction with the environment occurs until processing is completed. This distinction is also described as dynamic vs. static input (Davis, 1990). In dynamic applications, input continues to arrive during processing which affects the final outcome.

Reactive systems respond to and interact with their environment in a non-trivial way. Reactive systems contrast with batch systems which typically have a small number of well-defined inputs and no other interaction with their environment. The behavior of a

reactive system depends on the behavior of the environment (Ledru, 1991). Most systems in which the interface design is a large part of the system design are reactive systems.

One of my basic assumptions is that a functional artifact is intended to help the potential user(s) of the artifact to achieve certain goals (Freeman & Newell, 1971). In general, reactive systems are designed to allow the user to carry out certain plans which will achieve his/her goals. The development task, then, is to come up with an artifact which provides services which assist the artifact users to achieve certain goals.

A reactive system can be seen as a component of a composite system. Composite systems are those which include components which may be software, hardware and human (Fickas, Anderson, & Robinson, 1990). Specification complexity does not depend exclusively on the size of the resulting implementation, but also on the difficulty of predicting the interactions between the as-yet unbuilt artifact and the environment in which it will be placed. The ability to model the environment, including the actions of external agents, is a difficult but necessary part of representing the composite system.

External agents can play both positive and negative roles in the functioning of a system. Most goals are achieved through the cooperation of external agents and the designed artifact. In the library domain, the patron supplies some information about a book and the card catalog supplies its call number. The check-out operation generally involves actions by the patron, a clerk, and one or more mechanical or electronic devices.

While an analyst can only directly influence the functions of the artifact(s), she must take the actions of all of the actors into account. In particular, interface design deals with providing the information (and possibly motivation) required by each actor in carrying out their portion of the overall task. For example, recall notices remind patrons to return books and fines encourage them to do so.

These reactive systems may be faced with multiple conflicting requests, and the analyst must specify a satisfactory strategy for responding to those requests. For example, in

the library, more than one patron may want to check out the same book. Another example is an elevator, which has to provide service for passengers going in opposite directions, from any floor to any other floor. Resolving potential interactions is a key aspect of the specification engineering process.

### Context: Developing an Artifact

Software engineering and similar development processes can be viewed as the production of a series of models beginning with a completely world-oriented (application-oriented) model and progressing towards models that are more and more machine-oriented (implementation-oriented) (Greenspan, 1984). At the same time that the models are progressing from the application domain to the implementation domain, they are also progressing from a problem description to a solution description. Each phase of the artifact development process involves producing a model that is slightly closer to the final solution than the last. In addition, each phase of the process can be associated with a particular kind of reformulation. An overview of the development process is depicted in Figure 1.1.

The artifact development process can be divided into three primary sub-processes: defining the problem, designing the solution, and implementing the solution. Requirements engineering addresses the first of these sub-tasks: determining what problem the solution should solve (Thayer & Dorfman, 1990). Requirements engineering can be further divided into three sub-steps: modeling the application domain, formulating the requirements, and constructing the functional specification.

The key participants of the development process are:

- the client who is requesting the artifact;
- the requirements analyst, who constructs the functional specification;
- the designer, who designs the artifact to conform to the functional specification;
- the implementer, who builds the artifact following the design.

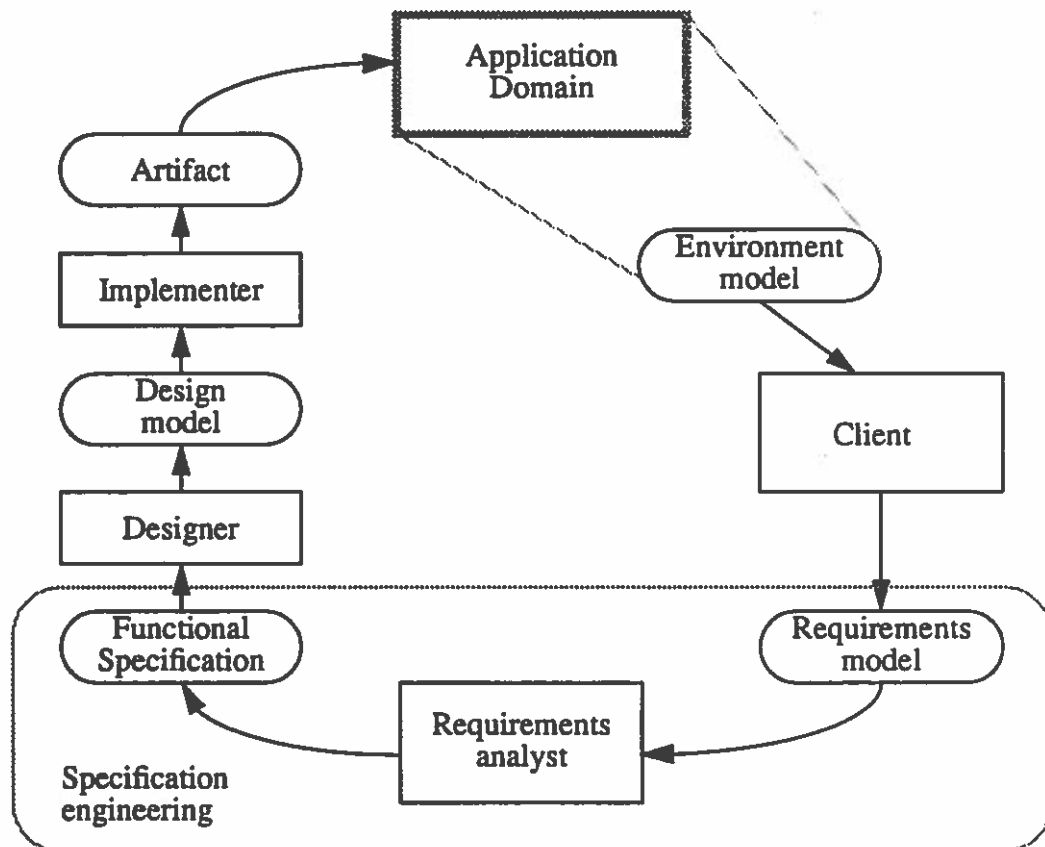


Figure 1.1 The development process begins with an application domain. A model of the domain is constructed and a client describes his requirements in terms of that model. The analyst composes a functional specification which satisfies the requirements. This is the portion of the development process referred to as specification engineering. Finally, the designer designs an artifact which satisfies the functional specification and the implementer constructs the artifact itself from the design.

First an environment model is built to abstract essential aspects of the application environment. This model describes objects found in the application environment, including a description of the properties of the objects and their behavior. The environment model describes relevant aspects of a portion of the world that encompasses the problem



situation, including relevant existing “systems” and their environment (Greenspan, 1984). The model of the application environment is constructed from descriptions provided by the client, textual descriptions, direct observation and participation by the analyst, and whatever other sources might be available.

A requirements model is built using elements in the environment model. Client requirements are expressed in terms of states and behaviors in the application domain, without regard for eventual implementation. The objects in the model should be application-domain concepts and not computer implementation concepts such as data structures. A good model can be understood and criticized by application experts who are not programmers (Rumbaugh, Blaha, Premerlani, Eddy, & Lorensen, 1991).

The client describes a set of requirements in terms of states and transitions between states. The client’s view of what constitutes desirable and undesirable states and behaviors are the client’s requirements. For example, a client describing the requirements for a library database might include checking out books as a desirable state transition and unauthorized access to a borrowing record as a transition to be avoided.

The next step is to compose a functional specification that satisfies the requirements. The analyst constructs an abstract description of the target artifact in terms of the services it provides to its users. The functional specification describes services needed in order to satisfy the client’s requirements. The analysis model is a concise, precise abstraction of what the desired artifact must do, not how it will be done.

The output of specification engineering is a description of the system to be constructed. The functional specification defines the external behavior of the target artifact (Davis, 1990).

The analyst and client evaluate the specification by considering ways in which the specified artifact might fail to meet the client’s requirements. Two types of failure might be detected: situations in which a desirable behavior is prevented, and situations in which

an undesirable behavior is allowed.

After validation, design decisions are made and details are added to the artifact model to describe and optimize the implementation of the specified services. The application-domain objects form the framework of the design model, but the artifact is implemented in terms of computer-domain objects. Finally the design model is implemented in a programming language, database, or hardware.

### Specification Engineering

In this dissertation I focus on the process of constructing the functional specification and ensuring that it matches the requirements. The initial description of the requirements provided by the client defines the tasks that must be performed, without saying how those tasks are to be accomplished. It is typically easiest for the client to state requirements in terms of application domain objects and actions, whereas a designer can best interpret requirements stated only in terms of artifact objects and services and an interface. Thus, an analyst's job is bridging the gap between a statement of client requirements, which might not mention the target artifact at all, and a specification of artifact services, which need not mention the application environment at all.

Requirements describe the ends to be achieved; the services that are available determine how easily the requirements can be met. The dialog between client and analyst corresponds roughly to the dialog between requirements and services. First the client states his requirements at a high level and the analyst points out potential conflicts and difficulties, based on her knowledge of possible services that could be provided by the artifact. The client then clarifies or revises his requirements and the analyst points out additional problems. Hopefully this process will eventually lead to convergence on a consistent set of requirements that can indeed be satisfied by a set of services that can be implemented in an artifact. "Design is a dialectic between goals - what is desired - and possibilities - what is

actually realizable” (Tong, 1988, p. 1).

The inputs and outputs of the specification engineering process are shown in Figure 1.2.

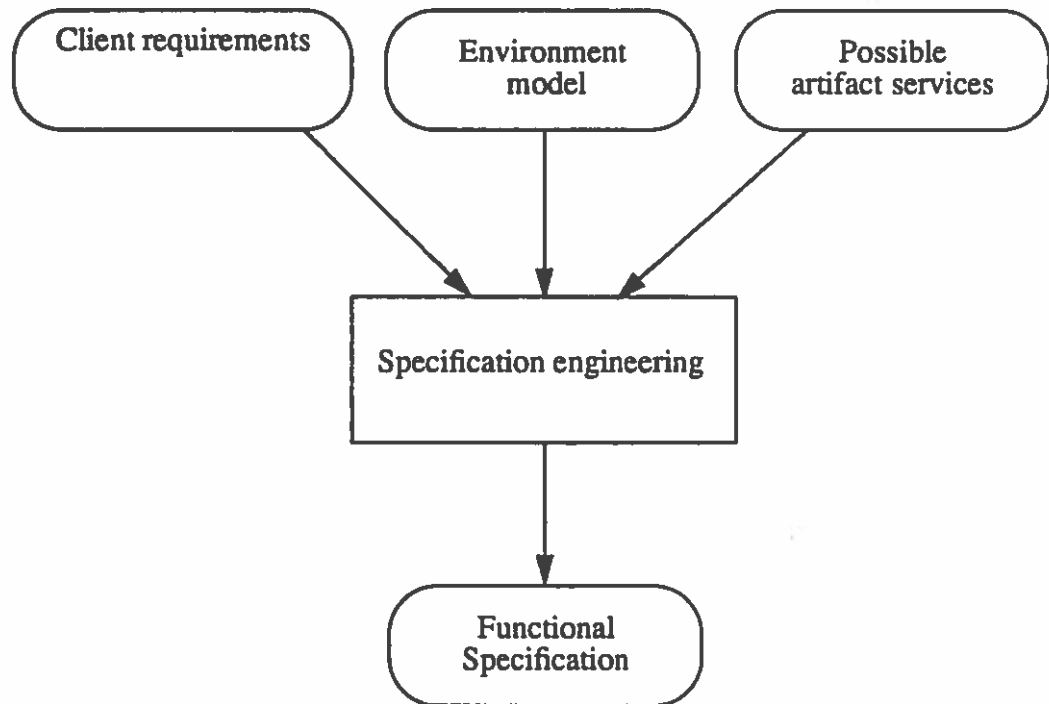


Figure 1.2 Inputs and outputs of the specification engineering process as formulated in the planning approach.

### Modeling to Predict Interactions between Environment and Artifact

Requirements engineering benefits from constructing abstract models of the proposed artifact. In the planning approach, the focus of the functional specification model is on the services provided by the artifact.

Modeling is a critical component of the requirements engineering process. Model

construction and evaluation provide one means for predicting the interactions between the target artifact and its environment. Understanding the interactions is necessary in cases where the interactions are likely to have an effect on the success or failure of the artifact.

Even with the best elicitation efforts, the requirements engineering process remains incomplete. The information gathered from interviews, observations, and direct participation provide information about the situation prior to the artifact's introduction, but not about the situation that will exist once the artifact is introduced.

In order to predict the interaction between the artifact and its environment, we need to place the model of the artifact in the model of the environment. The resulting model is not a functional specification, but is needed to interpret the functional specification (Greenspan, 1984).

#### Getting Feedback from Client

One reason for building intermediate models is to allow intermediate feedback from the client, before resources have been expended on developing an artifact which does not do what the client expects. Those decisions that significantly affect the role of the artifact user should be modelled in a way that the client can interpret and evaluate. In order to determine whether or not the behavior of the target artifact is appropriate it is necessary to also model the environment it is responding to. Capturing the interactions between the artifact and its environment involves reasoning about sequences of actions, the states they produce, and the reactions to those states.

#### Composite Model of Artifact in its Environment

The functional specification is used by the analyst as a model which helps in the understanding of the ramifications of certain design decisions. A model allows the analyst to test feasibility of ideas and to evaluate decisions prior to actually building the artifact.

Many researchers have proposed constructing conceptual models to aid understanding (Balzer, Cheatham & Green, 1983; Duggins, 1991; Greenspan, 1984). Due to the complexity of systems today, an additional layer of understanding is needed between the real world and the functional specification. Constructing a composite model assists the analyst in understanding the problem situation before the solution system is described.

The composite model is generally viewed as an abstract model of the application reality, i.e., the system within its environment. It describes the users' views of the context of the system, and typically identifies major user services, important relationships, and all external properties of the system. It depicts abstraction, assumptions, and constraints about the application and usually views the application in an extended time perspective. In short, the model forms the semantic basis for the contents of the system (Bubenko, 1980).

### Why Scenarios?

Client requirements are non-operational. They cannot be directly implemented, nor is there a direct means for determining whether a given functional specification supports them (Fickas & Nagarajan, 1988).

I view a requirements analyst's task as a kind of composition problem: from a given set of possible services, select a subset that can be used to satisfy a set of requirements. The analyst must understand what plans the user is likely to employ and specify artifact services which will accommodate those plans.

In constructing and validating a functional specification, the properties of primary interest are the services provided by the artifact. Scenarios provide a means of determining whether or not those services satisfy the client's requirements. A scenario represents a sequence of actions that transforms one state into another. The link between client requirements and specified services is made by analyzing sequences of actions to determine whether a particular transition between two states can possibly occur. If the transition is

desirable, then failure to find a plan is a deficiency in the functional specification. If the transition is undesirable, then finding a plan is a deficiency.

### Empirical Observations Show Analysts Using Scenarios

The context for this work began with empirical studies (Fickas, Collins, & Olivier, 1987) that revealed that analysts frequently used example scenarios in their discussions with clients. The scenarios were used both to confirm understanding, and to point out potential problems with the client's requirements.

The analyst's experience allows her to arrive at an illustrative scenario, or rule out the possibility of such a scenario much faster than would the client left on his own. However, once the scenario is described in terms of effects in the application domain, the client is easily convinced that a deficiency exists.

I decided to investigate the use of scenarios in requirements engineering in greater detail. As one means of investigation, I decided to construct a computer model of the specification engineering process, built around existing artificial intelligence planning techniques. The objectives of such a study include learning more about the role of scenarios in requirements engineering, proposing an initial model of the specification engineering process for evaluation and examination, and to show how scenarios could be constructed and manipulated automatically using existing work in AI planning.

### The Planning Approach

My goal is to model the process of using scenarios to compose and validate a specification. The validation process consists of two parts: first, ensuring that it is possible for the client requirements to be met, by finding scenarios in which the required tasks are completed. Second, uncovering deficiencies in the specification by showing scenarios in which undesirable states can be reached.

The planning approach takes a state-oriented view of requirements: clients express their objectives in terms of state transitions to be accomplished. A state transition describes a change from an initial state to a final state. Similarly, restrictions are expressed in terms of states: avoid transitions which result in states in which undesirable conditions hold.

In order to model reasoning about transitions between states, I employ planning techniques developed in the artificial intelligence community. The basic artificial intelligence planning task is to find a sequence of actions that will lead from an initial state to a goal state. The sequence of actions is composed by selecting actions from a predefined set. Actions are introduced which reduce the difference between the initial and goal states.

I view specification engineering as a similar sort of problem. The goal is to select a set of services from a predefined set. The task is to find the right combination of services such that client requirements are met and client restrictions are not violated. We are not looking for novel solutions in the sense that a new service must be defined. We are looking only at the problem of finding a workable combination of existing services. The novelty, if any, is finding a particular combination of services that had not been considered before.

In the planning approach, the knowledge that the analyst uses is encoded in the form of operators, together with the objects and relations manipulated by those operators. The analyst finds a set of operators that can be used to achieve the client's goals, and ensures that the preconditions of those operators can be established and maintained.

### Problem Domain

The test domain for my system is that of physical resource management systems, such as libraries. Objects in this domain are things like books, video tapes, patrons, customers, staff. Relations are things like item-borrowed, password-of. Typical operators are check-out, check-in, query-for-who-has-what-item. The domain knowledge we use is

taken from (1) texts and articles on analyzing problems in the domain, and (2) protocols of human analysts, familiar with the domain, constructing and critiquing specifications (Fickas et al., 1987).

### Example Problem: Library Privacy

An example will illustrate the types of problems the planning approach is intended to address. Suppose we are trying to come up with a functional specification of a library record-keeping system.

The first version of the specification contains a variety of services to be provided by the system, such as recording the checking out and returning of books, querying for various types of information, etc.

We can see how additional services might be added to the functional specification in the following example. Suppose one requirement is that borrowers have a way of being reminded of what resources they have checked out. One scenario is as follows:

borrower A checks out resource R;

A forgets the identity of R;

A queries the record-keeping system to find what resources A has checked out;  
the system displays the list of resources checked out to A, including R;

A is reminded that A has checked out R;

A checks resource R back in (eventually).

If borrowers are to be allowed access to their own borrowing records, then this plan or another like it must be made available. The “forgetting” and “reminding” actions are part of the environment model, and are therefore available regardless of what services are provided by the record-keeping system. The query operation, on the other hand, is a service provided by the record-keeping system. Therefore, the ability for a user to access their borrowing records is included in the functional specification as a service to be pro-



vided.

Next, suppose that the client wants to restrict the record-keeping system to prevent one user from finding out what resources another user has borrowed. In order to prevent users from violating other users' privacy, the query action has a precondition that users can only query on their own borrowing record.

The planner finds a plan which achieves the prohibited condition. Given an initial state of two borrowers A and B, and a resource R to be borrowed, the plan for reaching this prohibited condition is as follows:

borrower A checks out resource R;

B queries the system to find what resources A has checked out;

the system displays the list of resources checked out to A, including R;

B learns that A has checked out R.

We have an undesirable situation: a prohibited condition is shown to be achievable with a particular plan. Can we modify the plan to avoid the bad outcome from being reached?

One solution would be to simply remove the query operator from the specification, thus squashing the plan once and for all. However, it would then be necessary to find an alternative plan for satisfying the original requirement, that of allowing borrowers to be reminded of what resources they have checked out.

Another solution is to use a password scheme to prohibit access to someone else's records. This is the solution typically found in computer-based systems.

By reasoning about scenarios in which services participate to achieve either desired or undesired results, the analyst may eventually be able to find a combination of services that satisfies the client's requirements. If the analyst is successful, then the resulting functional specification can then be given to the designer for further refinement and implementation.

### Motivation for This Research

This research is aimed at modeling portions of the requirements engineering process. The resultant model has two benefits: first, as a means of understanding and discussing the requirements engineering process; and second, as a basis for building automated tools to support portions of the process.

The notion of “requirements engineering” is similar to that of “software engineering.” That is, a process that was once poorly understood and somewhat mystical is deemed critical enough that a concerted effort is made to understand and formalize the process. Once the process is understood, tools can be provided which serve to support it and reduce the burden on humans.

### Requirements Engineering is Important, Difficult, and Poorly Understood

The success of the entire project depends on the quality of the functional specification. It serves as the basis for communication among clients, users, designers, and implementors of the target artifact. The design and implementation must be verified against the specification (Yeh, Zave, Conn, & Cole, 1984).

The consequences of errors made in the requirements engineering phase carry throughout the entire software development process. Errors made in requirements and design are the most costly to detect and correct (Boehm, 1981). Errors introduced in the earliest phases of development take 1.5 to 3 times the effort of an implementation error to correct (Yeh et al., 1984). There is little advantage to building the system right, if you are not building the right system.

Unfortunately, the process of how requirements evolve from a client’s “wish list” to a formal specification of the target artifact is poorly understood and poorly supported by automated tools (Davis, 1990; Fickas & Nagarajan, 1988; Thayer & Dorfman, 1990).

### Problems Due to Poor Requirements Engineering

While demand for software is increasing, software production continues to be plagued by failures and cost overruns. A significant percentage of delivered software systems are completely unsatisfactory, extremely late, far over their budgets, and poorly suited for the intended users of the systems (Blum, 1992; Boehm, 1984; Davis, 1990; Duggins, 1991; Greenspan, 1984).

Typically, the reasons for these failures can be traced to inadequacies in the requirements analysis and specification phase of the software life-cycle. Poorly defined requirements are believed responsible for software project failures and for software that does not meet user needs. According to Blum, "Requirements analysis is the most important step in the entire software process. Most projects fail or exceed costs because this initial activity is not carried out properly" (Blum, 1992).

The problems associated with software development will only get worse as larger, more complex projects are attempted. One view is that the core challenge in requirements engineering is the analyst's need to deal with the large volume and wide diversity of knowledge associated with the requirements engineering process (Harris, Johnson, Benner, & Feather, 1992). This knowledge includes domain models, initial requirement conceptions, abstracted views of requirements, formal descriptions of systems, and stereotypical ways to modify these descriptions.

### Arguments for Automating Requirements Engineering

Requirements engineering is a worthwhile task to try to automate for several reasons. The benefits of automation include reduction in clerical errors, increased optimization, better documentation and reusable software (Balzer, Cheatham, & Green, 1983). Mechanizing design may increase productivity and reduce errors (Mostow, 1985). Auto-

mation provides an explicit record of assumptions and choices (Balzer, Goldman & Wile, 1978).

In very large systems, no one understands the whole system, so explicit representation is critical and automated support for accessing, querying and manipulating the models is needed (Devanbu, Ballard, Brachman, & Selfridge, 1991). Furthermore, large systems involve large quantities of diverse information, requiring knowledge-based systems rather than traditional tools (Harris et al., 1992).

The advantage of representing design methods as computer programs is that no information or process is hidden as “experience” or “judgment”. All of the representations and processes are explicit and open to review (Simon, 1981).

Many of the tasks associated with manipulating all of this information are relatively routine. If these routine tasks can be formally defined, they can also be automated. By providing automated tools to handle routine tasks, we leave programmers free to focus on more difficult parts of the problem (Rich & Waters, 1986).

Eventually, automated tools may help carry out portions of the design process. Automatic requirements engineering would help with the search for specifications. Maintenance at the requirements level rather than lower levels would reduce the large costs currently associated with maintaining large systems. Finally, incorporating sufficient knowledge into automated tools would allow their use by non-programmers (Steier, 1989).

### Requirements Engineering is Not Well Understood

The main body of knowledge of specification construction resides with human experts who use informal “seat of the pants” techniques, supplemented by a variety of guidelines and design languages used to record the outcome of whatever internal deliberations might be occurring. Practical methodologies that do exist supply notation and guide-

lines which can be interpreted by a human software analyst, but are not adequate for a formal model of the requirements engineering process (Fickas & Nagarajan, 1988). We do not fully understand the process we are trying to assist (Maarek & Berry, 1989).

Models will help us to understand the process of requirements engineering. The process of requirements engineering is not well understood, and is usually taught by example. As with other complex tasks, building a model of requirements engineering precise enough to automate should ultimately contribute to education and practice in the field (Steier, 1989).

As a means for understanding, computer models provides a way to develop and test ideas in addition to observation (e.g., Curtis, Krasner, & Iscoe, 1988; Lubars, Potts, & Richter, 1993) and controlled tests with experts and novices (e.g., Soloway & Ehrlich, 1984).

### Requirements Engineering is an Ill-structured Problem

Building an automated requirements engineering system can also drive progress in artificial intelligence. Requirements engineering is an ill-structured problem (Newell, 1969), since the exact form of the desired solution is unknown at the start of the problem. Many decisions must be made heuristically, on the basis of incomplete knowledge. Any robust specifier must make conjectures about appropriate design steps, test out those conjectures, and back up and try again if an initial guess proves incorrect. Although this kind of search among alternatives is a classic part of AI systems, it is still difficult to control such search when the relevant knowledge may come from several sources (Steier, 1989)

### Research Goals

The thesis proposed and investigated here is that techniques from artificial intelligence planning can be usefully applied to the problem of requirements engineering. Spe-

cifically, I am looking at requirements engineering for a class of artifacts known as reactive systems. Planning techniques are useful in composing functional specifications and analyzing those specifications with respect to safety requirements.

In addition to determining the extent of the role of planning in requirements engineering, I am also interested in identifying the remaining pieces: if one assumes that planning addresses part of the requirements engineering problem, what additional processes are required to complete the picture?

### Research Methodology

My approach to investigating the thesis was to implement a state-of-the-art planner, apply it to a specific requirements engineering problem, and analyze the strengths and weaknesses of the resultant process. The research project has involved designing and developing the prototype system, identifying interesting domains, developing the knowledge base for the domain, running tests, and evaluating the results. I provide an initial claim for feasibility and demonstrate how the system operates on a relatively small test problem. Finally, I evaluate the limitations and issues that arise in using the prototype and suggest directions for future research.

### Contributions

My primary contribution is the application of automated planning techniques to the construction and manipulation of scenarios in specification engineering. This work consists of several parts.

- A formulation of the requirements engineering problem that allows artificial intelligence planning techniques to be applied.
- An approach to the composition and analysis of specifications.
- An implementation of the planning approach.

- Evaluation of the implementation and the approach.

This dissertation presents one model of the requirements engineering process and discusses the merits and demerits of that model. The model is based on transferring and extending work in artificial intelligence planning to the domain of requirements engineering.

In this dissertation I make two primary claims. First, reasoning about scenarios is a useful way of composing and validating functional specifications. Second, existing planning techniques can be employed to construct and manipulate scenarios for use in specification engineering.

### Overview of the Dissertation

The dissertation has three major goals: first, I introduce one view of the requirements engineering task seen from a planning perspective. I introduce the terminology and definitions needed to formulate the requirements engineering problem so that a planner can be applied. Second, given this problem formulation, I describe how an automated planner can be used to generate scenarios which, in turn, are used to select functions to incorporate into the target artifact. Third, I evaluate the planning approach by applying it to a benchmark problem in the requirements engineering community, the library problem.

### Chapter 2: Problem Formulation

Chapter 2 presents the specification engineering problem formulated in such a way that it can be addressed by an automated planner.

A planning problem is expressed in terms of an initial state and a goal state. A solution is a sequence of actions which leads from the initial state to the goal state. The problem is solved by searching in the space of plans for a plan that solves the problem. In order to construct plans, a set of predefined operators is provided as input. New operators are

added to the plan in response to deficiencies detected. In a planning problem, a deficiency is a goal which is not achieved in the current plan.

A specification engineering problem can be expressed as a set of planning problems. The client describes a set of desirable states and behaviors and a set of undesirable states and behaviors. These are represented in terms of initial and final states. A solution is a set of services to be provided by the artifact being specified. The problem is solved by searching in the space of functional specifications for a set of services that satisfies the requirements. In order to construct functional specifications, a set of predefined services is provided as input. New services are added to the functional specification in response to deficiencies detected. A deficiency is a desirable state transition which cannot be achieved by the current functional specification. The search space is pruned by rejecting functional specifications which allow the violation of safety constraints.

### Chapter 3: Planning Approach to Specification Engineering

Chapter 3 gives an overview of the planning approach. It presents the high-level decision model used in the planning approach. The process model relies on search through a space of specifications. The search is driven by the detection and elimination of deficiencies in the current specification, and guided by the constraints imposed by the client's requirements and by selection rules that take those requirements into account.

In the planning approach to specification engineering, deficiencies are incompleteness and unsafeness. Incompleteness occurs when a desired behavior cannot be achieved with existing services in the functional specification. Unsafeness occurs when an undesired behavior can be achieved with services in the functional specification.

Addressing incompleteness can be formulated as a planning problem. The desired state transition is presented to the planner as an initial state and a final state. The planner attempts to construct a plan which leads from the initial state to the final state. If a plan is



found, those artifact services that contribute to the plan are included in the functional specification.

Detecting unsafeness can also be formulated as a planning problem. A state transition which represents a safety violation is presented to the planner as an initial and final state. If a plan is found, the functional specification fails to meet the client's requirements.

#### Chapter 4: Structured Knowledge

Chapter 4 is a more detailed description of my planner, OPIE. I describe how knowledge is represented and organized in OPIE so that decisions are made as intelligently as possible, but can be made using weak methods if selection knowledge is not available. I then describe how the techniques used to reduce search in planning can be extended for reducing search in specification engineering.

Any problem solving method that relies on search needs to be concerned with combinatorial explosion. Combinatorial explosion refers to the exponential growth in size of the search space relative to the depth. The size of the search space can be approximated as  $O(b^d)$ , where  $b$  is the branching factor (number of children at each node), and  $d$  is the depth of the search tree.

Structuring the knowledge base helps to reduce search. Using macro-operators shortens the depth of the tree for solutions which use the macro-operators (while increasing the branching factor and therefore making performance worse on other problems). Using generalized operators allows greater control over the branching factor, moving deterministic choices higher in the tree, reducing the overall branching factor within the tree. Abstract macros can be used to divide the problem into semi-independent sub-problems. While each sub-problem requires a search space which is  $O(b^d)$ , the size of the overall search space is the sum rather than the product of these search spaces. I then describe how knowledge-based approaches allow decisions to be made without search.

I describe how each of these techniques has been applied in the planning domain. I then describe how the techniques can be extended for reducing search in the space of functional specifications. I describe how the use of generalized operators is useful in repairing specifications with safety violations.

### Chapter 5: Results and Evaluation

I have implemented a state-of-the-art planner and applied it to a specific requirements engineering problem, the library problem. In chapter 5 I present the results of running the prototype system on the library problem.

My evaluation of the planning approach to specification engineering is divided into five parts. The first is expressiveness: while the use of preconditions and post-conditions to describe actions is important, it is not sufficient for expressing everything we might want to say about a functional specification. The second section discusses ambiguity issues. These issues were first raised in Wing's (1988) review of a dozen papers (IWSSD4, 1987) discussing alternative approaches to the library problem. The third section is also based on Wing's review. This section discusses issues that Wing referred to as incompleteness issues. The fourth section discusses limitations of the approach for guaranteeing a complete, correct functional specification. The fifth section discusses efficiency.

### Chapter 6: Conclusion

Chapter 6 summarizes the dissertation, presents my conclusions and discusses future work. The results of my experience indicate that planning is indeed a central component in requirements engineering. In addition, my experience also provides insight into those aspects of requirements engineering that cannot be accomplished by the planner, but which must interact with the planner to complete the overall requirements engineering process.

## CHAPTER II

### FORMULATING THE SPECIFICATION ENGINEERING PROBLEM

The purpose of this chapter is to describe the specification engineering problem from the perspective of artificial intelligence planning.

In order to better understand a complex process, it is useful to decompose that process into smaller pieces that can be examined individually. I propose a decomposition of the specification engineering process that includes planning as a central component. I explore this view by describing how planning fits into the overall process of requirements engineering.

A planning problem is expressed in terms of an initial state and a goal state. A solution is a sequence of actions which leads from the initial state to the goal state. The problem is solved by searching in the space of plans for a plan that solves the problem. In order to construct plans, a set of predefined operators is provided as input. Operators from this set are added to the plan in response to deficiencies detected. In a planning problem, a deficiency is a goal condition which is not achieved in the current plan. The search space is pruned by rejecting partial plans which contain a constraint violation.

A specification engineering problem can be expressed as a set of planning problems. The client describes a set of desirable states and behaviors and a set of undesirable states and behaviors. These are represented in terms of initial and final states. A solution is a set of services to be provided by the artifact being specified. The problem is solved by searching in the space of functional specifications for a set of services that satisfies the requirements. In order to construct functional specifications, a set of predefined services is provided as input. New services are added to the functional specification in response to

deficiencies detected. A deficiency is a desirable state transition which cannot be achieved by the current functional specification. The search space is pruned by rejecting functional specifications which allow the violation of safety constraints.

In the first section of the chapter I present a brief review of the traditional definitions of planning as used by the artificial intelligence community. I then present the specification engineering task and discuss it in terms of planning processes. I describe the inputs and outputs of specification engineering in planning terms. I then take a step back to look at what parts of the requirements engineering process have been included and what parts have been left out of the formulation.

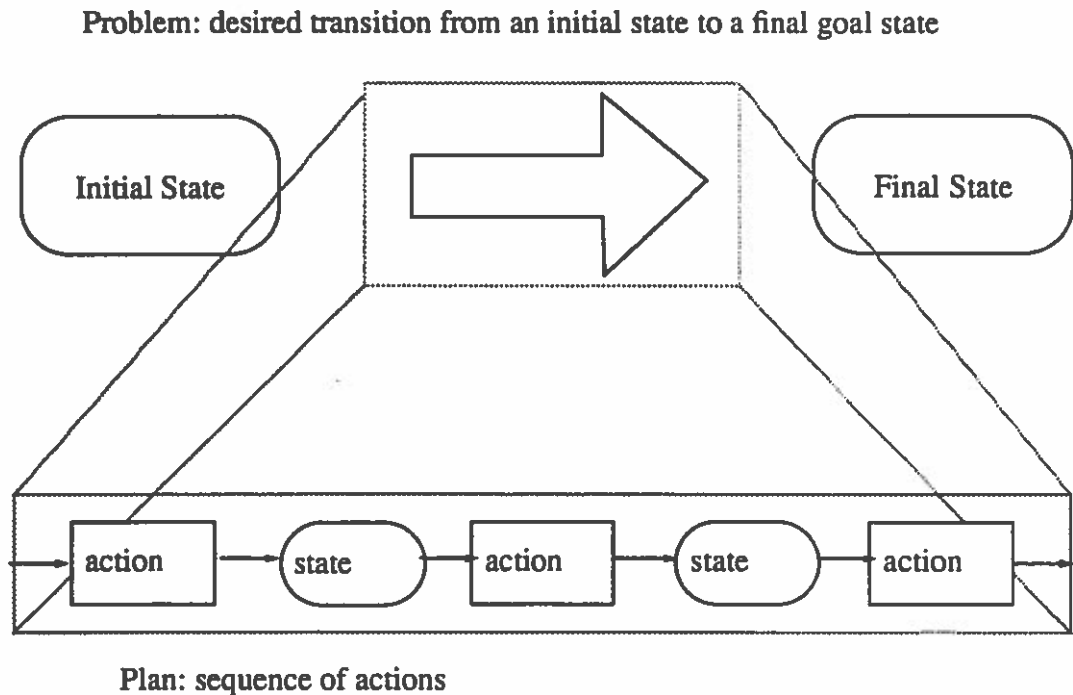
### Planning: Overview

The purpose of this section is to present a brief review of the planning concepts required to understand the point of view taken in the dissertation. This section presents the planning task, defines the terminology, and describes the process used to automatically compose a plan to achieve a given goal.

The view presented here is somewhat simplified. A more detailed description of the planner used in this work, called OPIE, is given in chapter 4. There are numerous sources of general information on the current state of artificial intelligence planning research, such as (Hendler, Tate, & Drummond, 1992; Allen, Hendler, & Tate, 1990; Genesereth & Nilsson, 1987).

### Classical Artificial Intelligence Planning Task

The input to a planner consists of an initial state description, a set of goal conditions, and a set of operators. The planner's task is to find a sequence of actions which will transform the initial state into a state which satisfies the goal conditions, as shown in Figure 2.1.



**Figure 2.1** The purpose of planning is to find a sequence of actions that lead from an initial state to a final or goal state.

### Definitions of Plan Elements

A plan is a model describing actions in the world that lead from one state to another. In order to describe the process of constructing a plan, I use a meta-model: a model used to describe another model. Entities, relationships, and actions in the world are represented by objects, persistences and operators in a plan, respectively. Objects, persistences and operators are plan elements manipulated by a planner. In this dissertation plan elements are given the same names as the things they represent, but use a different type face: a `book` is a plan element that represents a book.

A plan is a description of a set of actions, the entities affected by the actions, and the

relationships that are altered or required by the actions. A plan consists of objects, persistences, and operators. The operators in a plan may be partially or totally ordered.

Objects represent physical entities and, more generally, anything which can be acted upon. In the library domain which I will use to illustrate the planning approach, the object types include `books`, `id cards`, `patrons`, and `staff`. A special sub-class of objects is that of actors, which represent entities that are capable of initiating actions, such as `patrons` and `staff`.

A persistence represents a relationship between entities that holds over some continuous period of time (McDermott, 1982; Dean & McDermott, 1987). Persistences in the library domain include things like `available (book)`, `responsible (patron, book)`, `author-of (author, book)`. I treat persistences as individuals, making it possible to distinguish between different occurrences of the same relationship. For example, if Abigail checks out *Gone With the Wind*, then returns it, and later checks it out again, two separate persistences of `responsible (Abigail, Gone With the Wind)` are required.

Finally, an operator represents an action. An action occurs whenever the state of the world is changed. For example, `check-out (patron, book)`, `return (patron, book)`, `list-books-by-author (author)` are operators in the library domain.

Operators can describe actions of any granularity, from “pick up a book” to “get a college education.” Macro-operators, which are large-grained operators composed of smaller-grained operators, are treated like any other operators. The only difference is that a macro-operator can be decomposed, that is, replaced by its component operators.

The effects of actions are described in terms of the persistences which are produced and consumed by the operator. An action description also indicates the preconditions which must be true if the action is to be performed successfully, but are not actually altered by the action. These persistences are used by the operator.

For example, Figure 2.2 presents a graphical representation of a simple check-out operator.

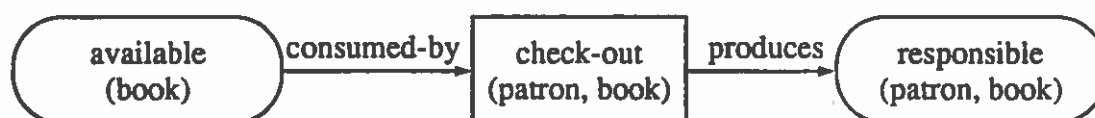


Figure 2.2 Example operator: check out, which consumes “available (book)” and produces “responsible (patron, book)”

‘Consumed-by’ and ‘produces’ are relations between plan elements (i.e., between an operator and a persistence). A relation between plan elements is referred to as a ‘meta-relation’ to distinguish it from a relation in the world described by a persistence. A meta-relation is part of a meta-model and describes a type of relationship between elements of a model.

An operator produces a persistence if the corresponding relation becomes true during the action. An operator consumes a persistence if the corresponding relation becomes false during the action. An operator uses a persistence if the corresponding relation must be true for the action to occur, is true throughout the action, and continues to be true after the action.

The input to the planning problem includes descriptions of types of entities, relations and actions for a particular domain. A description consists of a symbol plus all of the symbols directly linked to it. Such a description is called a schema. Schemas represent long-term knowledge that carries over between problem solving sessions. Schemas encode knowledge used in the model construction process. They are used as templates for creating new model elements and defining their relationships with other elements.

A state describes the world at a particular point in time. A state description can be derived from a plan description by collecting the set of persistences which were produced

prior to the time of interest and were consumed later.

The initial state of a plan is a set of persistences that are declared to hold when the plan begins. The goal is a set of persistences which are required to hold when the plan ends. That is, the goal persistences must either hold in the initial state or be produced in the plan.

In some cases a planning problem can include a set of global path constraints that must be satisfied in order for a solution to be acceptable. A path constraint is essentially a predicate on a partial solution sequence, rather than on a single state or operator. For instance, a path constraint may disallow particular subsequences of operators, or require that the plan be completed in a specified amount of time (Carbonell, 1983).

Figure 2.3 summarizes the planning problem. The inputs are a description of a desired transition between an initial and final state and a set of operator schemas that indicate the possible actions that can be included in a plan. The planner attempts to find some sequence of operators from the input set that will lead from the initial state to the final state.

### The Planning Process

The process of finding a sequence of operators that achieves the desired transition is a constraint satisfaction problem. Constraint satisfaction problems take the form “find a point  $x$  in a space  $X$  such that  $x$  satisfies the constraints  $C_i(x)$  and maximizes an objective function  $F(x)$ ” (Freeman and Newell, 1971, p. 621). Solving constraint satisfaction problems is an incremental search process.

Planning can be viewed as search in a space of plans for a plan which leads from the initial state to the final state, as shown in Figure 2.4. The root node is the null plan, i.e., do nothing.



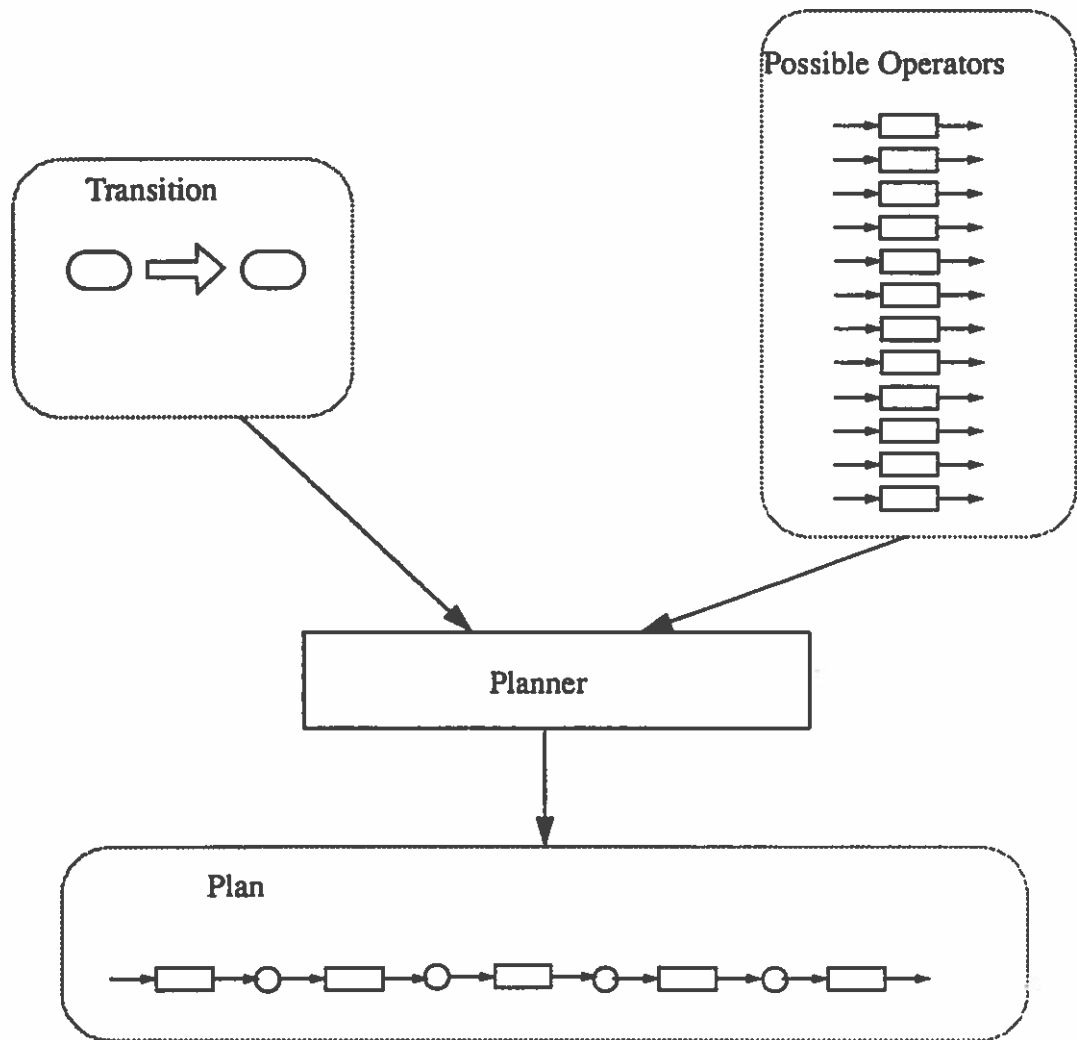


Figure 2.3 Inputs and outputs of the planning process.

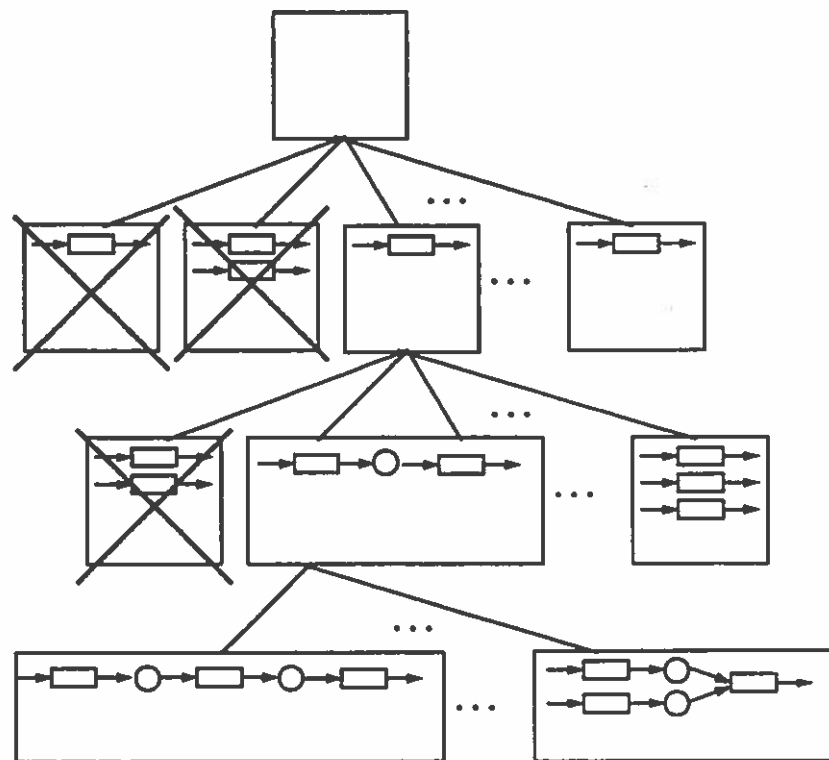


Figure 2.4 Each node in the search space represents a different sequence of actions.

### Planning is a Configuration Problem

Planning can be viewed as belonging to a class of problems known as configuration problems (Mittal & Frayman, 1987). A configuration problem is one of specifying a set of components and the interconnections among them such that the configuration satisfies a set of requirements.

The input for a configuration problem consists of: (A) a fixed, pre-defined catalog of component schemas, where a component is described in terms of its required and possible connections to other components; and (B) some description of the desired configuration.

Originally the term “configuration problem” was used to refer to configuring physical components into structures. Documented systems configure computers, networks, operating systems, buildings, circuit boards, keyboards, printing presses, and trucks (Mittal & Frayman, 1987). In planning, operators and persistences are first-order elements and can be “configured” as well. The connections between elements are the produces, consumes and uses links between operators and persistences. The requirements to be satisfied are that the goal persistences and all operator preconditions are produced, either by operators in the plan or by the initial state.

Planning requires a knowledge base of object, persistence, and operator schemas and some means of accessing and assembling the relevant plan element descriptions given a problem description.

#### Persistences Without Producers are Deficiencies

We use the notion of slots and fillers to drive the planning process. Goal persistences begin with an empty slot for a producer.

Persistences with empty producer slots play a special role in planning. These unproduced persistences represent deficiencies of the current plan: relations which need to be achieved, but are not yet attributed to a particular producer. The primary task of a planner is to find a producer for each unproduced persistence.

In order to repair a deficiency, a planner must have a way of determining possible fillers for slots. This requires storing information about the possible associations among objects, relations and actions in a knowledge base.

Deficiencies (i.e., empty slots) are the primary retrieval index into the operator set. For example, suppose we are trying to achieve the goal `can-use (patron, book)`. We use pattern-matching on `can-use (patron, book)` to access the corresponding persistence schema in the knowledge base. We then follow the `produced-by` link to find possible

producers of that schema. Each producer is a potential filler for the slot. Suppose we have stored the knowledge that check-out is an operator that produces can-use (patron, book). Check-out becomes a candidate for inclusion in the plan.

When an operator is added to the plan, its preconditions become additional deficiencies which the plan must address. For example, checking out a book requires locating the book, which may in turn require that the patron have access to a catalog of books in the library. Each of the preconditions must be linked to a producer. That producer may have preconditions as well. This results in a chaining process leading backward from the goal to the initial state. The process of identifying differences and selecting operators to reduce them is called means-ends analysis (Nilsson, 1980). A solution is found when all persistences in the final state and all operator preconditions are either produced by an operator in the plan or are contained in the initial state.

### Specification Engineering

According to the planning approach which I am proposing, the output of the specification engineering process is a functional specification. The primary input to the process is the client's statement of requirements, which defines the problem to be solved. Additional input comes from knowledge about the problem domain. Domain knowledge includes knowledge about the environment into which the artifact will be placed, and knowledge about what kinds of artifacts can be built. In particular, we are interested in the actions that can be performed by actors in the environment, and the services which might be provided by the artifact being specified. These inputs and outputs are shown in Figure 2.5.

I next describe the process used to construct a functional specification given these inputs and outputs.

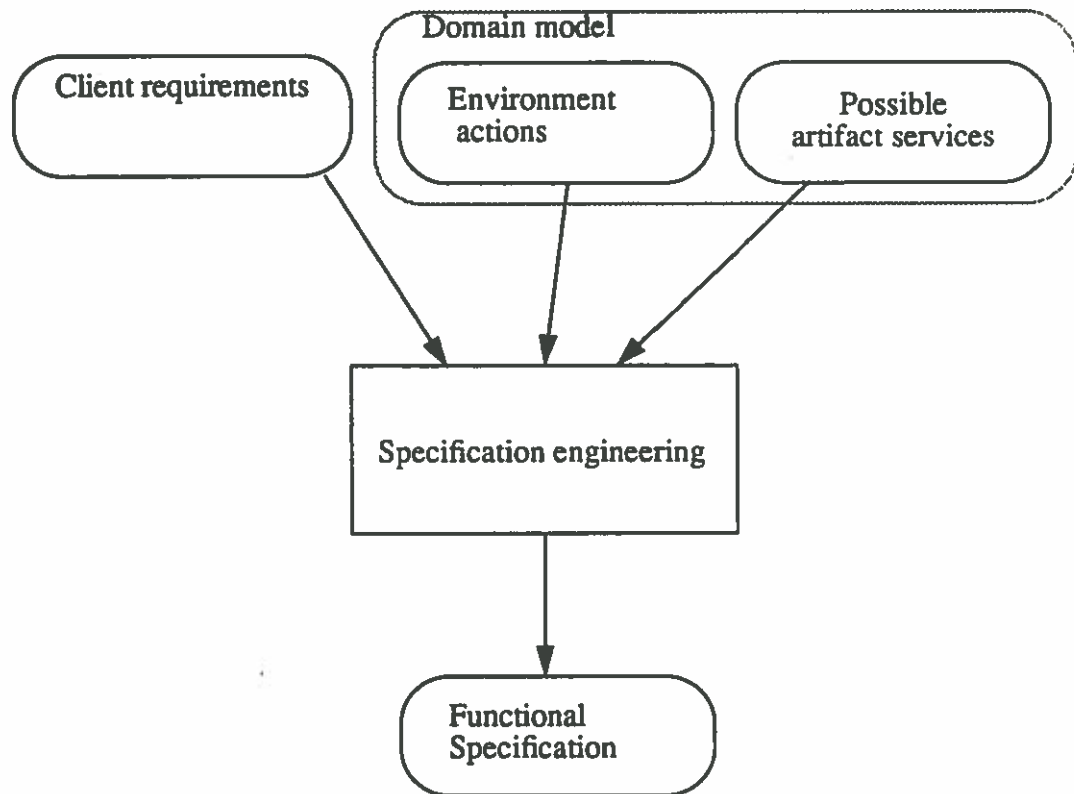
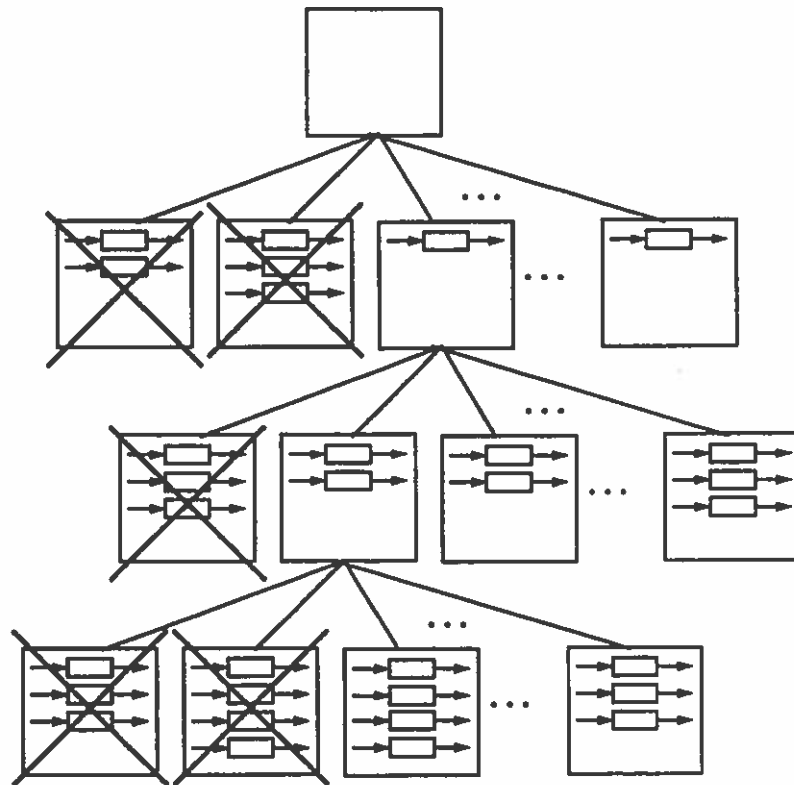


Figure 2.5 Inputs and outputs of the specification engineering process.

### Search for a Functional Specification

Like planning, the process of determining which set of services satisfy the client's requirements is a constraint satisfaction problem. Specification engineering can be viewed as search in a space of functional specifications for a specification consistent with the client's requirements, as shown in Figure 2.6. The root node may be the null specification, i.e., an empty set of services, or it may be a partial specification.



**Figure 2.6** Each node in the search space is a functional specification defining a different set of services.

We need to consider two issues: how do we choose which set of services to consider at each node of the search space, and how do we determine whether or not that set of services satisfies the requirements?

In the worst case, the search would involve repeatedly taking an arbitrary set of services and determining whether or not that set satisfies the requirements.

### Specification Engineering is a Configuration Problem

Specification engineering can be viewed as being closely related to the class of con-

figuration problems. The specification engineering task is to identify the required services of a target artifact. The inputs include requirements which the target artifact must satisfy and a large set of possible services. The output is a functional specification of a set of services which the target artifact must provide.

The requirements analyst tries to select the best combination of services to satisfy all of the requirements. For example, there might be a variety of ways to check out a book; the analyst would select the method that best fits with the other services to satisfy the requirements.

A similar view of specification engineering can be found in D'Ippolito & Plinta (1987) and in Goedicke, Schumann, & Cramer, (1989). A similar approach is taken in the Requirements Apprentice (Reubenstein, 1990; Reubenstein & Waters, 1991). Kramer discusses an approach that treats software design as a configuration problem (Kramer, Magee, & Sloman, 1989).

In the remainder of this chapter I describe the specification engineering problem in greater detail, showing how the inputs and outputs can be described in planning terminology. In the next section I describe the requirements. In the following section I discuss the output, a functional specification. Then I discuss the domain knowledge needed to construct a functional specification that satisfies the requirements.

### Client Requirements

In the planning view, the specification engineering process is one of determining what set of services best satisfies the client's needs.

One input is a set of requirements which the target artifact must satisfy. Requirements identify behaviors which are either required or prohibited in the target context.

Clients typically formulate requirements in terms of tasks to be accomplished (Fikas and Nagarajan, 1988). Clients often say "I need a (better) way of getting from point A

to point B”, where “point A” and “point B” refer to states of the world. “Getting from state A to state B” can be expressed formally as a desired transition between an initial state and a final or goal state.



Figure 2.7 Requirements can be expressed as state transitions.

The desired transitions described by clients generally represent a large class of similar tasks. For example, obtaining a book from a library is a generic transition that describes a large class of specific tasks involving many different patrons and many different books. The initial state is one in which a patron wants access to a book and the book is available in the library; the goal state is one in which the patron has the use of the book.

Figure 2.8 presents an example set of requirements for the library domain. This description was used at the 4th International Workshop on Software Specification and Design as a common problem for researchers to illustrate their specification languages and methodologies.



**The library problem (TWSSD4, 1987):**

Consider a small library database with the following transactions:

- 1- Check out a copy of a book / Return a copy of a book;
- 2- Add a copy of a book to / Remove a copy of a book from the library;
- 3- Get the list of books by a particular author or in a particular subject area;
- 4- Find out the list of books currently checked out by a particular borrower
- 5- Find out what borrower last checked out a particular copy of a book.

There are two types of users: staff users and ordinary borrowers. Transactions 1,2,4, and 5 are restricted to staff users, except that ordinary borrowers can perform transaction 4 to find out the list of books currently borrowed by themselves. The data base must also satisfy the following constraints:

- 1- All copies in the library must be available for checkout or be checked out.
- 2- No copy of the book may be both available and checked out at the same time.
- 3- A borrower may not have more than a predefined number of books checked out at one time.

Figure 2.8 Initial statement of requirements for the library problem.

**Safety Requirements (Restrictions)**

In addition to describing state transitions that should be enabled or made easier, requirements may include state transitions that should be prevented or made difficult. The analyst is not only concerned with helping potential artifact users achieve their goals, but also with preventing certain undesirable conditions.

For example, books should not be stolen or mis-shelved. Furthermore, revealing information about a patron may constitute invasion of privacy. A client might add "do not allow an unauthorized patron to access another patron's borrowing records" as a restric-

tion on the target system. Although not stated explicitly, this restriction is contained in the requirements in Figure 2.8.

The analyst must anticipate user actions that, accidentally or intentionally, might result in an unacceptable state. To the extent possible, opportunities for those actions must be eliminated. The analyst must deal with both errorful and irresponsible behavior

### Performance Constraints

Finally, requirements include non-functional preferences such as “minimize cost” and “maximize reliability”. These preferences are used to rank alternative designs. Resource limitations are a source of constraints on the set of permissible services. As an example, what would be the cost of having a staff person oversee (or actually carry out) all queries to a library’s borrowing records as a means of avoiding violations of patrons’ privacy?

Functional requirements establish the essential behaviors for all correct implementations. Nonfunctional requirements establish operational properties for the delivered implementation (e.g., response time, storage constraints, ease of use) (Blum, 1992; Roman, 1985).

### Goal: Balance Functionality, Safety and Cost

Broadly speaking, the specification engineering process is one of finding a balance between functionality, safety, and cost. While in the ideal case we would like to enforce all of the client’s requirements, the cost of enforcing a requirement can often be greater than the cost of allowing the requirement to be violated. Explicit representation of the cost of the operators and plans used in satisfying a requirement is needed in order to evaluate the cost of enforcing that requirement.

The task is to ensure that the desired effects can in fact be achieved and the undes-

ired effects cannot be achieved, while at the same time keeping the cost of satisfying the requirements as low as possible. There are frequently conflicts between giving users what they want and stopping them from carrying out undesirable behavior or over-running available resources. In addition, the ideal solution is often very expensive but an alternative exists which is much cheaper but less satisfying. Part of getting the specification right is determining what trade-offs between different requirements, and between requirements and costs, are acceptable to the client.

Our challenge as software engineers is to identify the functional requirements clearly and verify that no decisions violate them, and to document the nonfunctional requirements and choose design alternatives that have the highest probability of being able to meet them (Blum, 1992).

In this dissertation the focus is primarily on functional requirements. In order to simplify dealing with non-functional requirements, I treat all non-functional requirements as being collected in a single cost or “utility” value. This value is used in choosing between two options that meet the same functional requirements: the option with the greater utility value is preferred.

### Representation of Requirements

A client’s requirements are expressed in terms of states and transitions between states in the application domain. These requirements fall into two primary classes: transitions which result in desired states and transitions which result in undesired states. I refer to these as achievement and safety requirements, respectively.

#### Achievement Requirements

An achievement requirement is very similar to a traditional planning problem: from a given initial state, reach a state in which a desired persistence is true. For example, the

requirements model for a library might include the transition shown in Figure 2.9 as an achievement requirement. This transition says that it should be possible for a book which is available in the library to be obtained by a patron, who is then responsible for the book until it is returned.

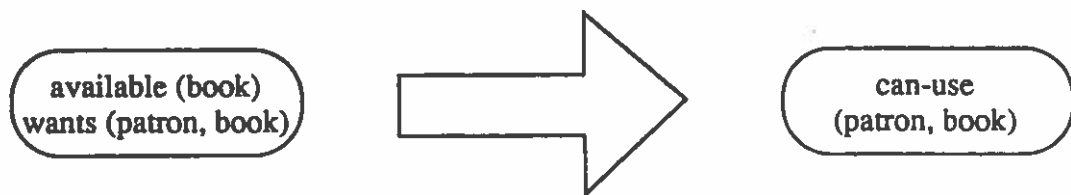


Figure 2.9 An achievement requirement: if a book is available in the library, then it should be possible for a patron who wants that book to obtain it.

### Safety Requirements

Safety requirements describe transitions that either consume states which should be maintained or produce states which should be avoided. A requirement to maintain a particular state can be expressed as a transition to be prevented: any transition in which a persistence to be maintained is consumed should be avoided. For example, a library might require that reference books remain in the library.

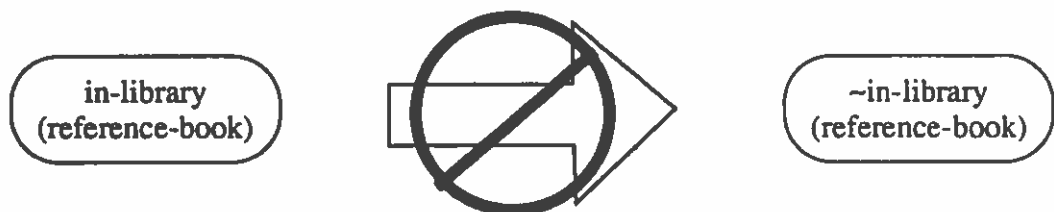


Figure 2.10 A state to be maintained: reference books should remain in the library. This can be represented as a safety requirement: avoid transitions that result in reference books leaving the library.

Similarly, the client may want to prohibit certain states from occurring. In this case, the requirement can be expressed as a transition in which an undesirable persistence is produced. For example, a library might want to avoid stolen books. Transitions in which states to be maintained are consumed and transitions in which states to be avoided are produced are prohibited transitions.

It is important for an artifact to provide services which enable transitions contained in the achievement requirements while preventing prohibited transitions. The goal of specification engineering is to identify a set of artifact services which simultaneously satisfy all of the client's requirements.

Thus, in the planning approach, requirements are initially expressed in terms of the state transitions that an artifact is intended or expected to affect, both positive and negative. This could be viewed as expressing requirements in terms of a set of test cases that the artifact is required to satisfy. The artifact meets the requirements exactly if it passes all of the test cases.

### Library Requirements

Figure 2.11 presents a state transition representation of the requirements in the library problem.

### Output: Functional Specification

In the planning view of specification engineering, a functional specification is a description of the services that must be provided by the target artifact if the client requirements are to be satisfied (Greenspan, 1984). Requirements are defined by the IEEE as "the capabilities necessary to satisfy a need" (Davis, 1990). A functional specification defines the inputs to the target artifact and defines and constrains the outputs of the artifact.

Services are represented as STRIPS-style operators, that is, in terms of their pre- and

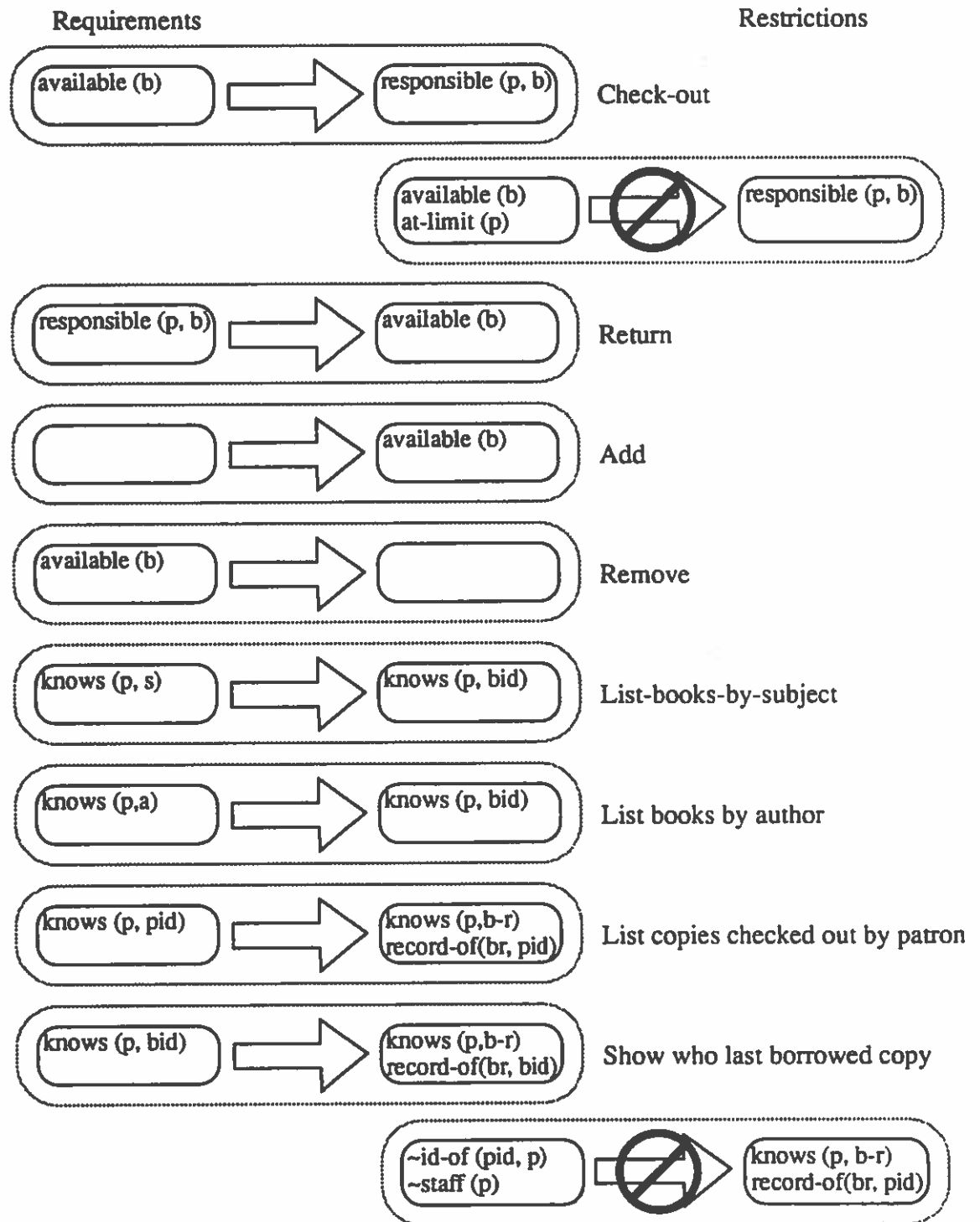


Figure 2.11 Requirements for the library problem

post-conditions. A functional specification, then, is represented as a set of operators to be implemented in the target artifact. Figure 2.12 presents a set of operators representing some of the services to be provided by a library record-keeping system. The services include storing, displaying and deleting records pertaining to borrowing transactions.

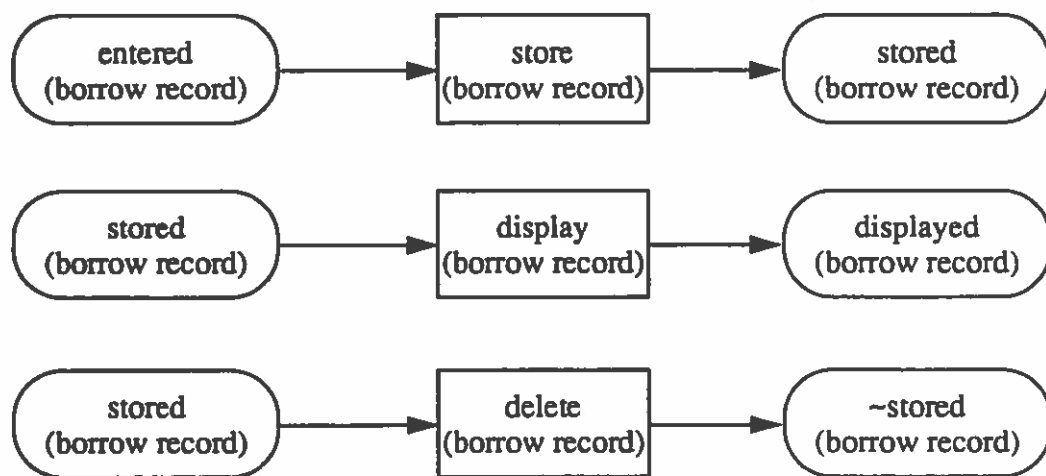


Figure 2.12 Example operators representing services which have been selected to be provided by a target artifact. These operators make up the functional specification of the artifact.

### Domain Model

Requirements engineering requires a significant body of information about the domain of the system being required (e.g., general information about libraries, patrons, and books). Since this information is shared by all of the systems in a given domain, it is a potential target for reuse.

### Potential Artifact Services

In specifying the behavior of an artifact, a key input is knowledge of the possible behaviors of this class of artifacts. The analyst must know something about how the artifact will be constructed and what the range of possible implementations might be. Without this knowledge, it would be impossible to ensure that the specified system could actually be built.

In most mature engineering domains, a central part of the design process is selecting appropriate components from a pre-defined set. The selection process may occur at a variety of levels. For example, in electrical engineering a component may be a single transistor, a chip containing many transistors, a printed circuit board containing many chips, or a complete system containing many printed circuit boards.

In specification engineering, the “components” to be assembled are the services to be provided by the target artifact. Services refer to those functions of the target artifact which respond to and affect conditions in the environment. These are the external, as opposed to internal, functions of the artifact. They represent the visible behaviors of the artifact, which are of concern to the client. The internal functions which are used to produce the visible behaviors are not of interest during the specification engineering phase of development. They will come into play during the design phase.

The domain model contains pre-defined operators which represent the set of services that could potentially be incorporated into the target artifact. Typical examples in software systems are services which store information, retrieve it, and modify it. As with components in other engineering disciplines, services provided by functional artifacts come in a variety of grain sizes. For example, editing a document or deleting a single character are services of an editor, at two different levels of granularity.

The representation for possible artifact services is the same as the representation of



services in a functional specification as presented in Figure 2.12.

### Environment Actions

Understanding the problem means modeling the environment. It is more or less accepted that the target artifact cannot be modeled independently of the environment in which it resides (e.g., the particular library in which a record-keeping system is to be placed). Indeed the boundary between the environment and system may not be fully defined until later stages of software development. The input to the specification engineering process must provide sufficient knowledge about the environment for the boundary to be precisely positioned and the interface to be defined (Finkelstein and Waters, 1989).

The knowledge base includes operators which describe the actions of external agents in the domain, which I call environment actions. This set includes the normal things that people do every day such as move from place to place, pick objects up and put them down, and push them about. Environment actions would be available whether or not the artifact is built.

Figure 2.13 presents examples of the kinds of actions that are available in the library environment.

### Summary: Defining the Specification Engineering Task

In this chapter I have presented a planning-oriented formulation of the specification engineering task: from a set of requirements, construct a functional specification.

In summary, the input to the specification engineering process is a set of requirements, expressed in terms of desirable and undesirable state transitions. The output is a set of services that make up the functional specification of the target artifact, represented as operators. The purpose of specification engineering is to specify a set of services which satisfy all of the requirements.

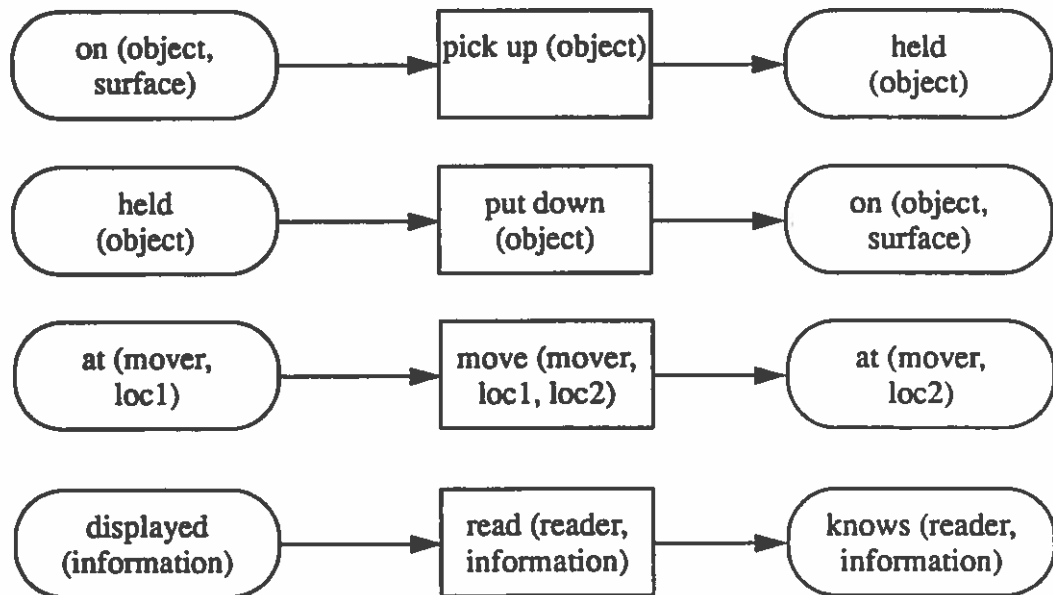
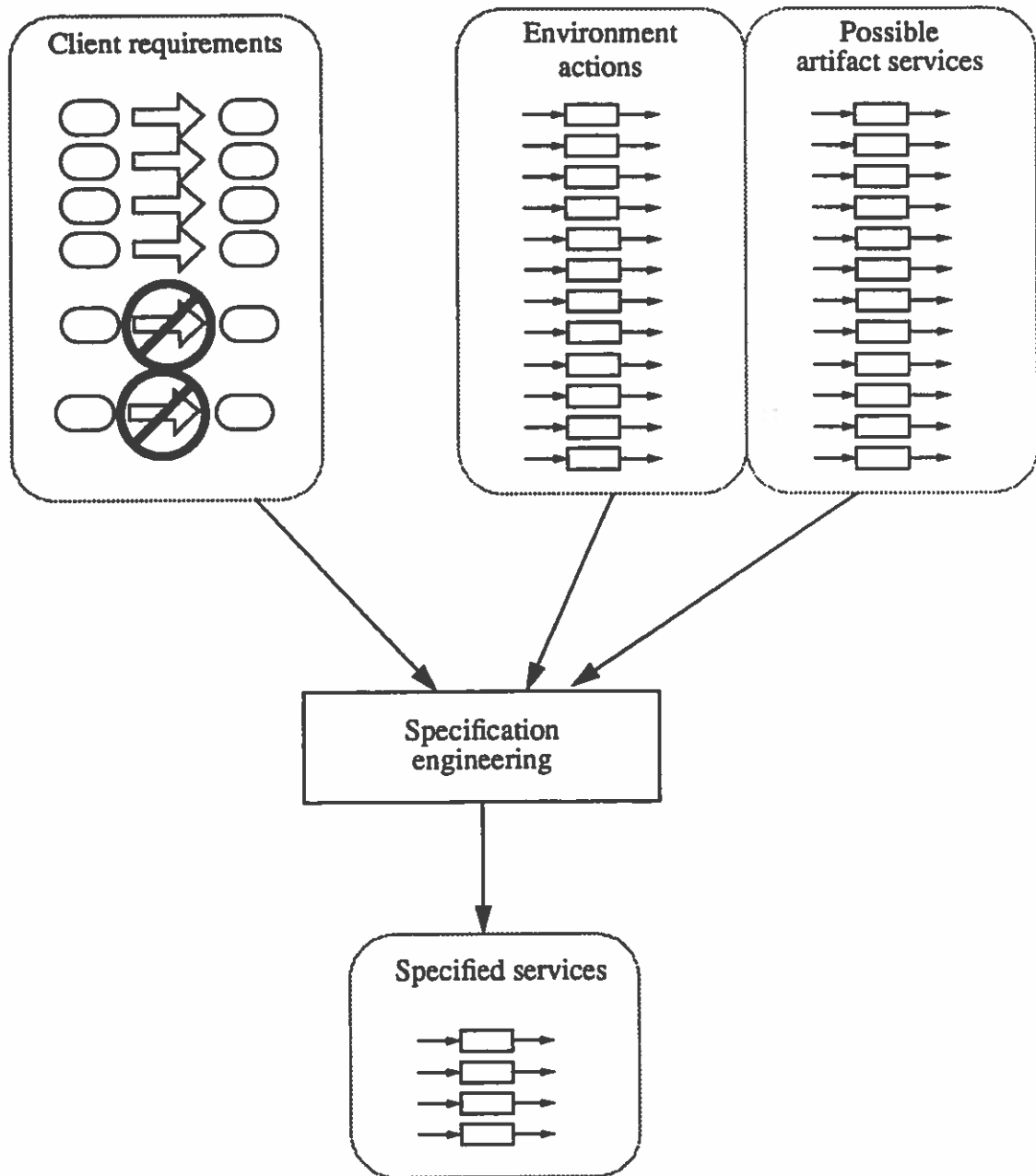


Figure 2.13 Example operators representing actions of actors in the environment in which the target artifact is to be placed.

Figure 2.14 presents an overview of the specification engineering process, as defined in this chapter.

The key points to be taken from this chapter:

- A central part of any design process is selecting components from a predefined set. In constructing a functional specification, the “components” to be selected are the services of the artifact. From a large set of possible services, select a subset that satisfies the requirements. The selected services become the functional specification of the target artifact.
- In order to determine what the necessary services should be, the analyst has to reason about the client’s enterprise-wide objectives. However, the services to be provided by the artifact should be expressed in terms of inputs and outputs of the



**Figure 2.14** Overview of the specification engineering process: inputs and outputs. The goal is to find a subset of the possible artifact services that satisfies the client requirements.

**target artifact:** when a user does  $x$ , the response should be  $y$ . Every such  $x$  and  $y$  pair represents a service to be provided by the target artifact. The task is to take a description of a client's needs, expressed in terms of the client's environment and experiences, and replace it with a description of a solution system, expressed entirely in terms of characteristics of the solution system.

- The requirements analyst must find a set of services that enable users to achieve their objectives but do not allow users to bring about unacceptable conditions. This tension between desirable and undesirable behaviors is a central issue in the planning approach to specification engineering.

### Specification Can be Seen as Planning Process

We now have a definition of a specification problem and criteria for a satisfying solution. Given that one is willing to live under these assumptions and definitions, I argue that construction of a functional specification can be viewed at least partly as a planning process. The next chapter describes how planning is used as a central part of specification engineering.

## CHAPTER III

### THE PLANNING APPROACH TO SPECIFICATION ENGINEERING

In this chapter, I describe the planning approach to specification engineering. The approach uses artificial intelligence planning techniques for composing and analyzing functional specifications.

In order for the analyst to ensure that the services are consistent with the behaviors desired by the client, the analyst must reason about the effects of the actions allowed by the services.

I view the problem solved by requirements analysts as being similar to that solved by planners. In planning, the input is in the form of a problem, which can be expressed as a transition between an initial state and a final state. The output is a sequence of operators that can be used to solve the problem. In specification engineering, the input is actually a set of planning problems, and the output is not a sequence of actions, but a set of services to be provided by the target artifact. However, the necessary services can be identified by first constructing plans that solve the individual problems.

In the next section I give a more detailed description of the specification engineering process and show how it can be decomposed into a set of planning tasks. These planning tasks fall into two categories: state transitions which are desired and state transitions which are prohibited. In section 3 I describe how an initial functional specification is composed by addressing desired state transitions. I then describe how the resultant functional specification is critiqued with respect to the prohibited transitions. If problems are detected during the critiquing phase, the specification must be modified to address these problems. The planner itself is not able to make these modifications directly. In section 5 I describe ways the planner might be extended to address these problems.

### Decomposing a Specification Engineering Problem into Planning Tasks

In this section, I show how the specification engineering task can be decomposed into a set of planning problems. As in any approach to problem solving that involves decomposition, we must show how the decomposition is accomplished, how the sub-problems are solved, and how the results are integrated into a solution to the overall solution.

#### Deficiency Driven Problem Solving

Both planning and specification engineering belong to a class of problems in which search is driven by finding and repairing deficiencies in the current partial solution. The general solution to this class of problems is to represent, detect, and eliminate deficiencies [IBIS]. This is similar to the deficiency-driven model of algorithm design proposed by Steier and Kant [TSE85]. The generic algorithm for solving this class of problems is shown in Figure 3.1.

The process begins with a deficient solution model. A model is deficient with respect to a particular set of requirements if there are problem requirements that are not satisfied in the model.

The first step is to identify deficiencies in the model and select one to address. The next step is to repair the deficiency. This involves accessing possible repairs, evaluating the alternatives, and selecting one. The repair is then applied to transform the model into a new model.

Repairing one deficiency may introduce new deficiencies, or restrict the possible repairs for another deficiency due to interactions between the repairs. These restrictions are expressed in the form of constraints on the final solution.

Once the constraints have been introduced, the model is evaluated for constraint violations which result from unforeseen interactions among components. If constraint viola-

tions are found, the solution model must be revised by retracting a previous commitment and making an alternative choice.

### Algorithm

- 1) Identify deficiencies
  - 2) Choose a deficiency to address
  - 3) Access possible repairs
  - 4) Evaluate repairs
  - 5) Select repair
  - 6) Apply repair
  - 7) Propagate constraints - prune incompatible repairs for other deficiencies
  - 8) Evaluate resultant model - may need to backtrack / explore alternative branches of search space
- Go to 1

Figure 3.1 Algorithm for solving problems using deficiency driven techniques.

### Deficiency Driven Specification Engineering

Deficiency driven specification engineering is an incremental process of generating a functional specification by addressing individual deficiencies of the current specification.

The search is carried out by repeatedly detecting and correcting deficiencies as shown in Figure 3.2. Each move in the search space begins with an analysis process which reveals deficiencies in the current specification, i.e., ways in which the specification fails to meet the client's requirements. The next step is to propose alternative repairs for a deficiency, and select a repair to be applied. Deficiencies are addressed by adding and remov-

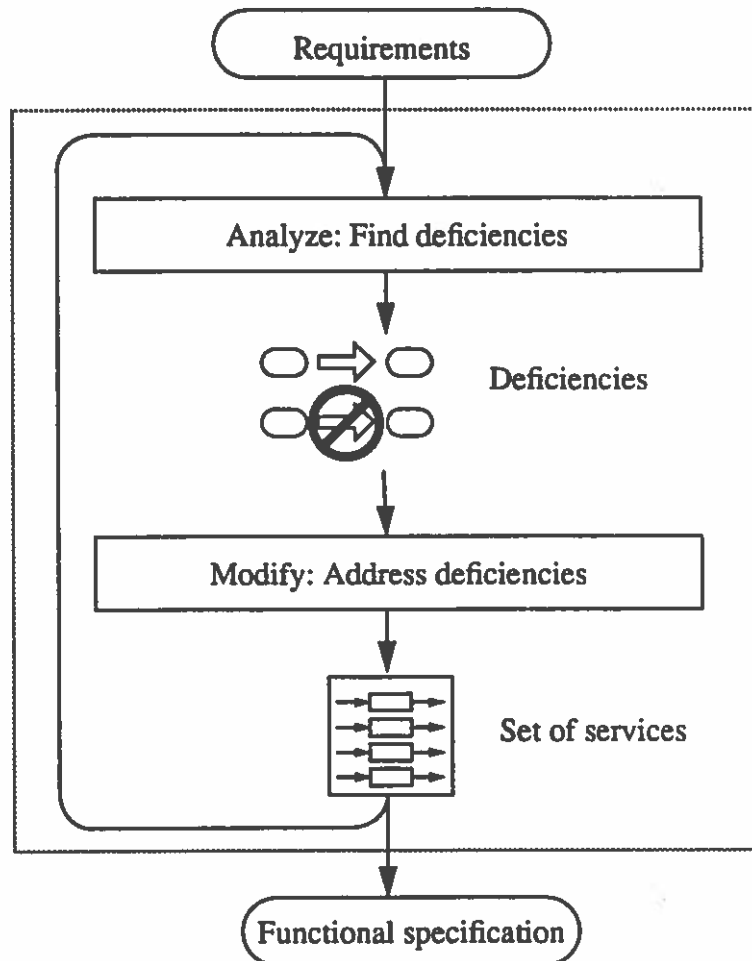


Figure 3.2 Deficiency driven specification engineering involves a cycle of detecting deficiencies and modifying the set of services to address the deficiency.

ing services to and from the functional specification. Applying the repair generates a new functional specification node in the search space. If all deficiencies are eventually eliminated, the resulting specification is a solution.

#### Types of Deficiencies in Specification Engineering

To satisfy the requirements, the services included in the functional specification must achieve the desired behaviors, must not achieve the prohibited behaviors, and must



be implementable given available resources.

A functional specification is said to be deficient when a requirement can be violated. In specification engineering, a deficiency is a mismatch between requirements and specification: a desired transition which cannot be achieved given the specified services, or a prohibited transition which can be achieved given the specified services. These two kinds of mismatches correspond to two broad classes of deficiencies: incompleteness and unsafe-ness [Fickas]. A functional specification is considered incomplete if a desired state transition cannot be accomplished. It is considered unsafe if a prohibited state can be achieved.

### Incompleteness

Incompleteness is a type of deficiency that occurs when a desired transition cannot be achieved (from an accessible state) using operators in the specification. For example, if patrons should be allowed to check out books but the functional specification does not contain any means for doing so, the specification is incomplete.

Figure 3.3 shows the requirement that a patron be able to obtain a book from the library.

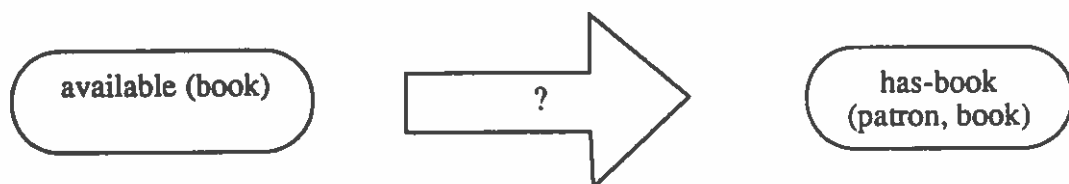


Figure 3.3 A desired state transition which cannot be achieved is a completeness deficiency.

Incompleteness is the basic deficiency in most modeling problems. For example, a traditional artificial intelligence planning problem is a model of a desired transition, missing the action(s) required to achieve the transition.

### Unsafeness

Unsafeness occurs when a prohibited transition can be achieved using operators in the specification. For example, if patrons should not be allowed to access other patrons' borrowing records, but the current functional specification can be shown to allow it, the specification is unsafe.

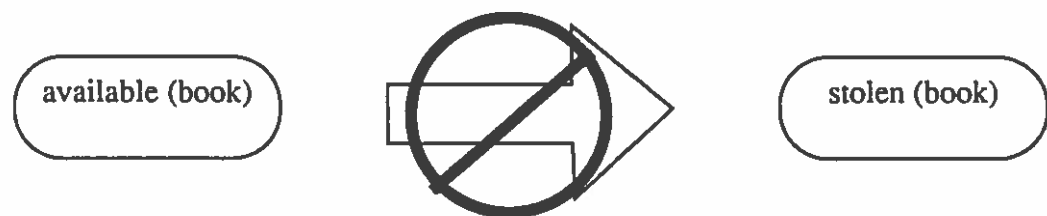


Figure 3.4 A prohibited state transition which can be achieved is a safety deficiency.

It is not only important for an artifact to enable certain goals of potential users; it is also often important to disable the goals of potential mis-users. It is important that the analyst have knowledge of potentially harmful plans that should be prevented or at least made difficult to achieve.

### Address Incompleteness and Unsafeness

The goal of specification engineering is to ensure that for every achievement requirement, a sequence of operators which accomplishes the transition must be available given the available services and environment actions. At the same time, no sequence of operators which accomplishes a prohibited transition should be available.

Incompleteness drives the search forward, adding new services to the specification. Unsafeness serves to limit the range of acceptable solutions. Achievement requirements are treated as primary requirements to be achieved, and safety requirements are treated as

secondary requirements which influence the choice of methods for achieving the primary requirements.

### Use of Scenarios

In order to detect and correct deficiencies in functional specifications, we reason about scenarios. A scenario is a sequence of actions showing how a transition from one state to another might occur (as illustrated in Figure 3.5).

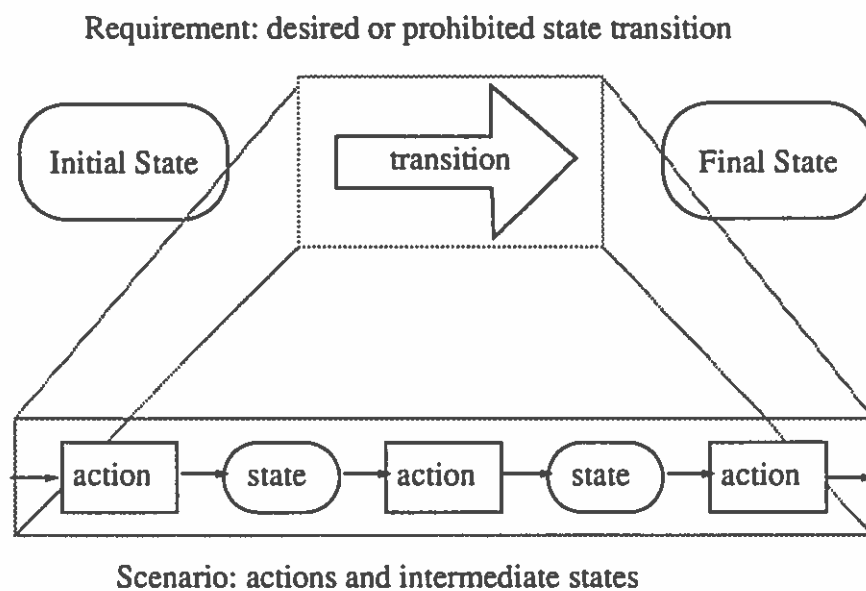


Figure 3.5 The relationship between requirements and scenarios. Scenarios show how state transitions can or cannot occur.

The possible actions in a scenario are determined by the services of the target artifact and the actions of the agents in its environment.

If a combination of environment operators and artifact services can be found which

accomplish a desired transition, that requirement is satisfied. On the other hand, if a sequence of environment and artifact operators can be found which achieve a prohibited transition, that requirement is violated. Scenarios serve to bridge the gap between requirements expressed in terms of states of the world and services provided by the target artifact.

Given this representation, we can use an automated planner to determine whether or not a given set of services allows the achievement of a particular transition. Finding action sequences which accomplish a given state change is exactly the standard artificial intelligence planning problem. The knowledge required is causal information linking actions to their preconditions and effects. The only difference between a scenario and a plan is that the analyst does not intend to execute the scenario. The analyst simply uses scenarios to predict possible events that the artifact will be involved in.

Knowing what transitions can and cannot be achieved tells us whether or not the functional specification satisfies the client's requirements.

#### Summary: Scenarios Bridge the Gap between Requirements and Services

A functional specification describes the services that an artifact should provide. Constructing a functional specification involves finding services which allow desirable events and disallow undesirable events. In order to bridge the gap between requirements and services, the analyst must reason about the effects of actions.

The approach to specification engineering proposed in this dissertation is based on the use of planning as a means of detecting and eliminating deficiencies in a partial specification. My approach uses planning to composing the functional specification by finding services that will enable artifact users to achieve their goals. It also uses plans to critiquing the functional specification by determining whether the services provided will have undesirable applications.

In the next section of the chapter I describe how OPIE, an automated planner, can be

used to detect and correct incompleteness in a functional specification. The following section discusses how OPIE is used to detect unsafeness in a functional specification. Although OPIE is not able to correct safety violations directly, section 4 describes ways in which artificial intelligence problem solving techniques can be applied to modify a specification found to be unsafe.

### Composition

The first phase in composing a functional specification is selecting operators which achieve the desired transitions in the client's requirements. The inputs to this process include the requirements plus a catalog of possible artifact operators and environment operators, as seen in Figure 3.6.

The functional specification is composed by planning. The set of operators required to accomplish the desired transitions will include both environment operators and artifact services. The artifact services make up the functional specification of the artifact. Thus, the planner selects a subset of the possible artifact services for actual implementation in the target artifact.

### Detecting and Addressing Incompleteness

Achievement requirements drive the initial operator selection process. First, OPIE ensures that every goal state mentioned in an achievement requirement can actually be achieved. For each transition mentioned in an achievement requirement, OPIE is given the task of finding a plan to achieve the transition. If the planner is successful, there exists a method for achieving the goal.

The analyst selects a desired transition from the requirements and asks OPIE to find a plan showing how that transition can occur. For example, consider the requirement from

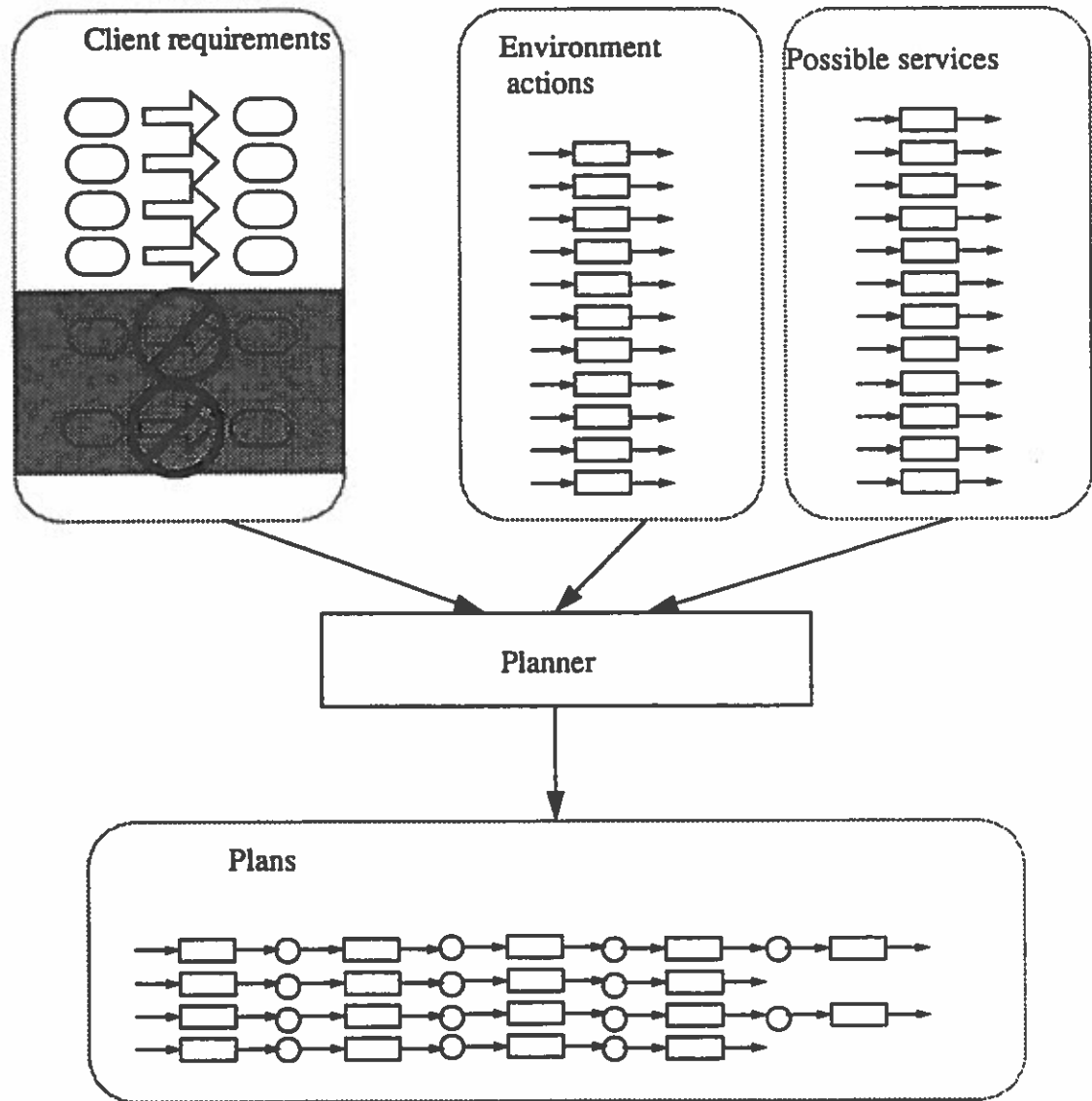
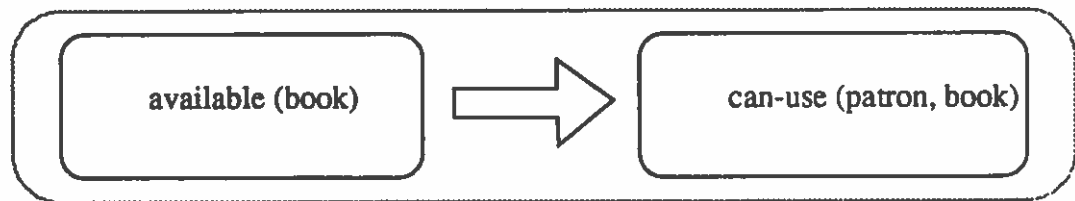


Figure 3.6 Composition phase: ignoring safety requirements, the planner finds plans in which a combination of environment and artifact operators achieve desired transitions.

the library problem that patrons be able to obtain books, shown in Figure 3.7..



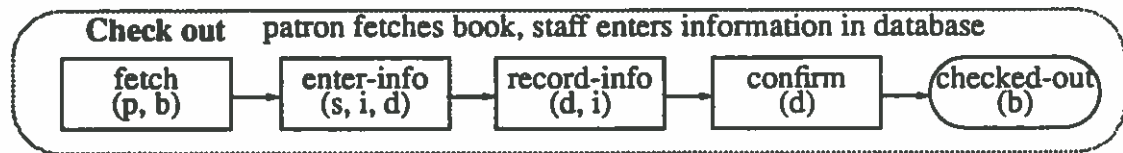
**Figure 3.7** Example achievement requirement from the library problem. The planner is given this transition as a planning problem. The planner looks for a sequence of environment and artifact operators to achieve the transition.

To construct a plan, the planner is given the set of possible artifact services plus the complete set of environment actions as its operator set. Together, these two sets of operators represent the composite domain model of the possible implementations of the artifact and its environment. Using a composite model allows an analyst to reason about the interactions between the artifact and the environment in which it is used.

OPIE searches in the space of plans consisting of any combination of environment and artifact operators. It tries to find a sequence of operators that achieves the state transition from the book being available in the library to the patron having possession of the book. This is a standard artificial intelligence planning problem and we omit the details of how the search is carried out. One plan for accomplishing the transition is shown in Figure 3.8.

#### Plans Involve Both Artifact and Environment Actions

Some of the actions in the plan will be performed by agents in the environment, others by the artifact being specified. For example, in the check-out plan in Figure 3.8, the patron selects a book and brings it to the counter, a staff person enters identifying informa-



**Actions > Conditions**

patron enters library

> patron in library

patron looks up book in catalog

> patron knows location of book

patron finds book on shelf

> patron has book

patron takes book to counter

> patron and book at counter

staff enters book-id and patron-id

> book-id and patron-id entered

database creates borrow-record

> patron responsible for book

patron takes book out of library

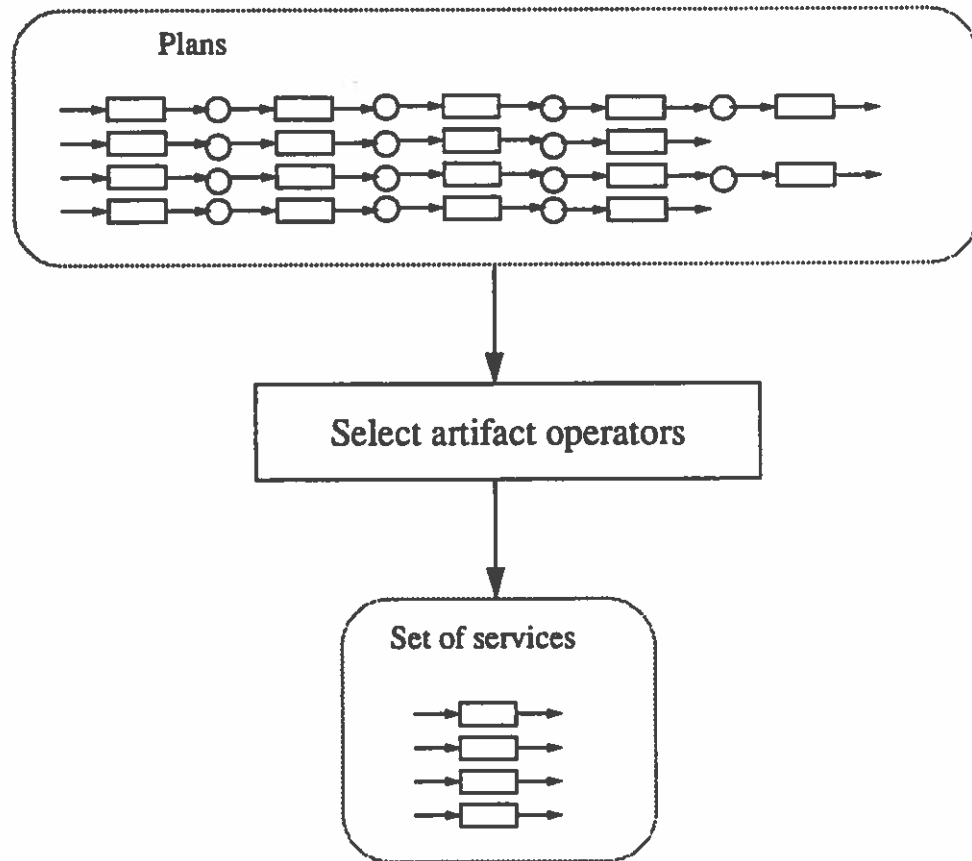
> patron can use book

Figure 3.8 Example plan for checking out a book. The plan includes a combination of environment and artifact operators which achieve the transition.

tion about the book and the patron, and the database creates and stores a record of the transaction.

Any artifact operator in the plan becomes a part of the initial functional specification as shown in Figure 3.9. Essentially, the requirements analyst has identified a particular way in which the required transition can be achieved. The artifact will be required to provide services such that the analyst's solution can be carried out.





**Figure 3.9** Those operators in the plans that represent artifact services are placed into the functional specification.

Typically there are a variety of plans which achieve any particular transition. Figure 3.10 shows some of the plans for checking out a book. All of these plans satisfy the requirement that patrons be able to obtain books from the library. However, it is not enough to simply find an acceptable plan for each requirement in isolation. It is necessary to coordinate the services so that all requirements are satisfied by a single set of services. The next section describes how interactions between achievement and safety requirements are used to eliminate some possible artifact services from consideration.

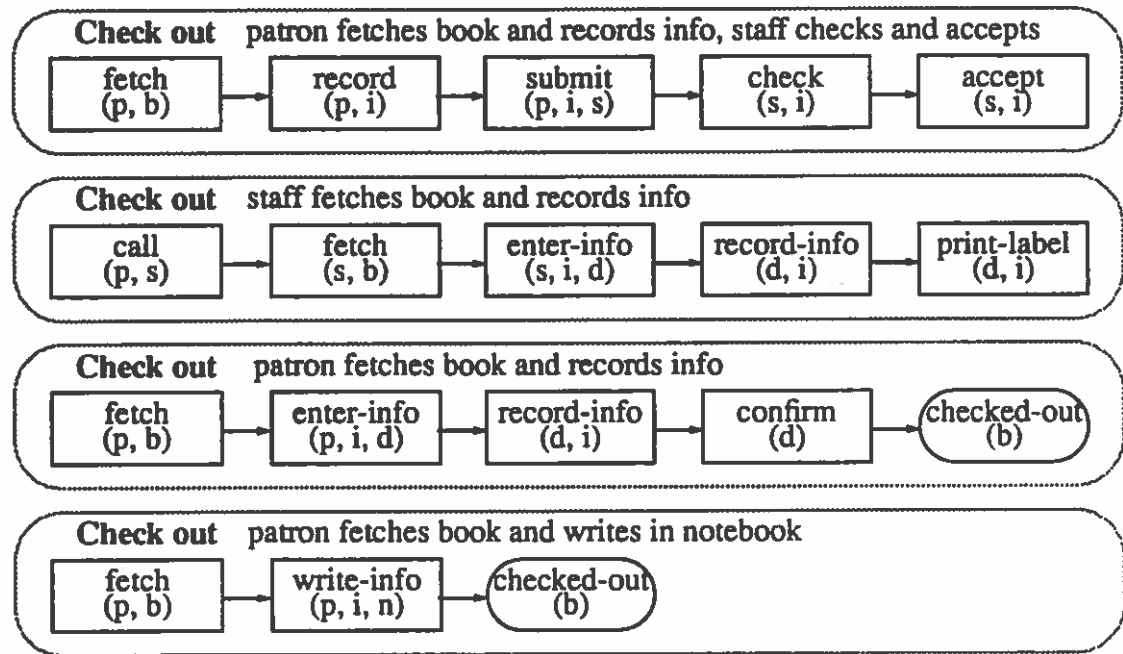


Figure 3.10 Representative plans that might be produced for achieving the check-out transition.

### Detect Safety Violations

The next phase of the specification engineering process is to critique the specification. While incompleteness drives problem solving forward, unsafeness serves to limit the range of acceptable solutions. The specification must be evaluated to ensure that none of the safety requirements are violated. The inputs and outputs of the critiquing phase are shown in Figure 3.11.

OPIE works as a devil's advocate, trying to poke holes in the specification by finding plans which achieve states outlawed by a safety requirement. For each prohibited transition in the client's requirements, the planner is given the task of finding a plan that violates the requirement.

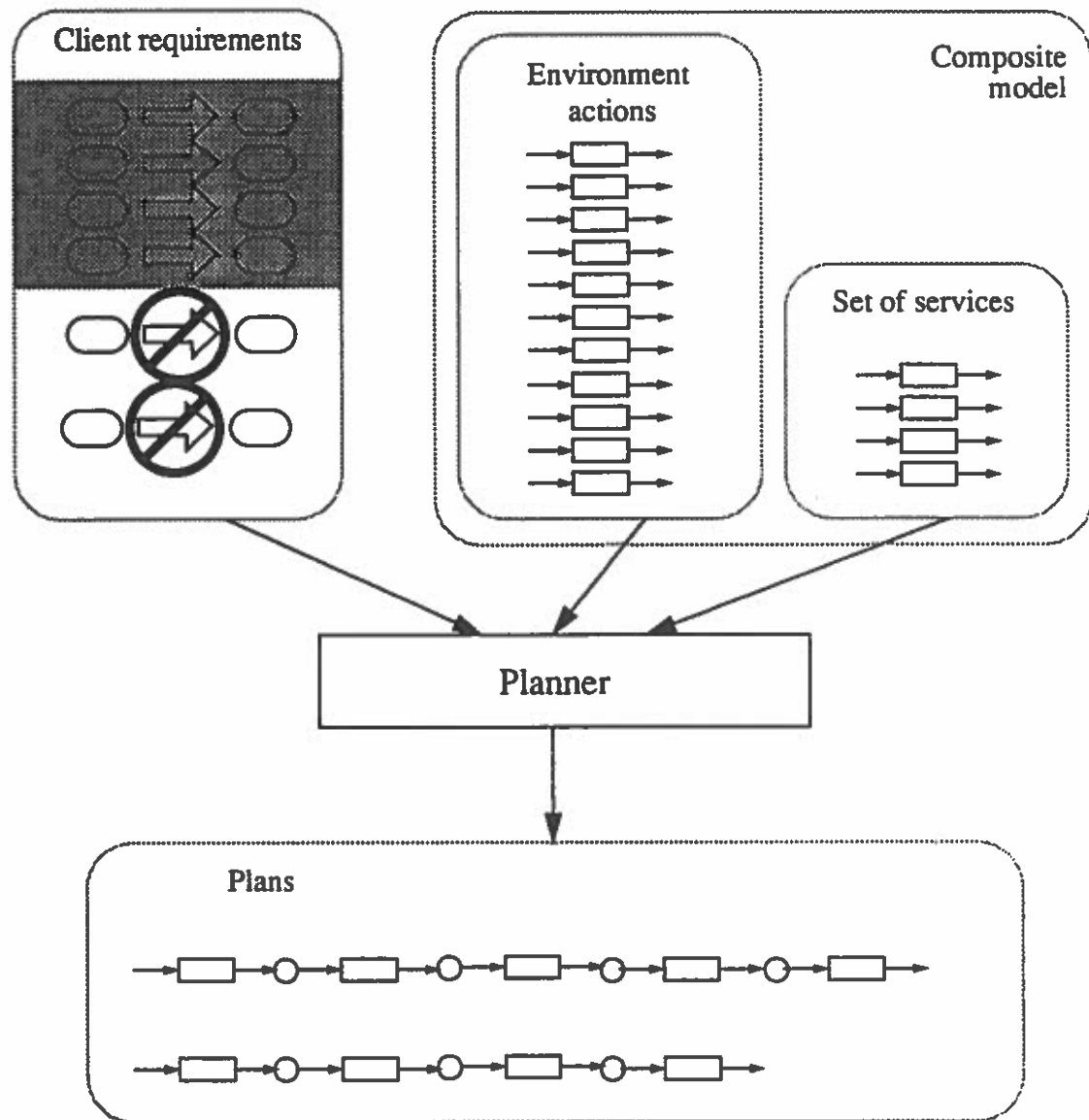


Figure 3.11 During the critiquing phase, achievement requirements are ignored. The planner works as a devil's advocate, trying to find plans in which safety requirements are violated. If such plans are found, the specification must be revised.

Each of the prohibited conditions is treated as a goal to be achieved. In this way, the set of safety requirements can be treated as a set of traditional planning problems. OPIE

searches for plans in which a safety requirement is violated. For example, Figure 3.12 shows a safety requirement that says that patrons must not check out more books than the limit set by the library.

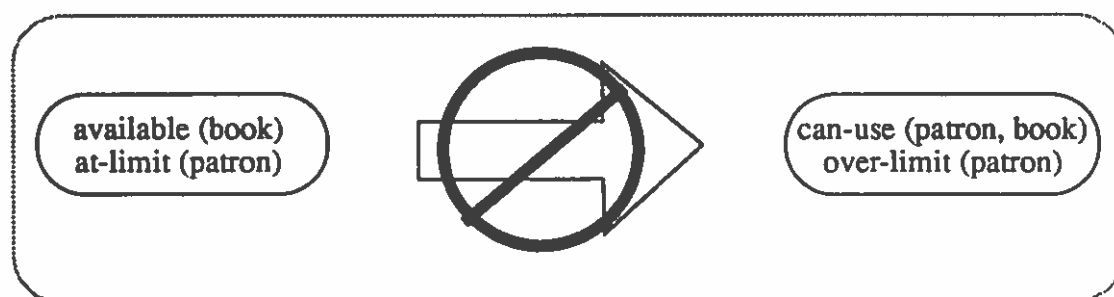


Figure 3.12 Example safety requirement for the library problem. A patron should not be able to check out a book if they have already reached their borrowing limit.

The inputs to the planner are changed slightly during the critiquing phase. Instead of considering all possible artifact operators, the planner only considers those artifact operators actually selected for the functional specification. We still take all environment operators into account, however. This combination represents the composite model of the proposed artifact and its environment.

If OPIE succeeds in finding a plan that achieves a prohibited condition, it has demonstrated by counter-example that the safety requirement is not satisfied.

For example, one requirement in the library problem is that a patron cannot check out more than a predefined number of books. OPIE would try to find a plan where a patron checks out more than the allowed number of books, thus showing that the specification is not consistent with the requirements. The same plan shown in Figure 3.8 that shows a patron obtaining a book can be modified slightly to show a patron who is at her borrowing

limit checking out a book and going over her limit. This is because the plan contains no check to see whether or not the patron is at or over their borrowing limit..

Once a safety violation is detected, the functional specification must be modified. This process is discussed in the next section.

### Address Safety Violations

Once safety violations are detected, the planner reaches its limit in terms of its ability to directly assist in the specification engineering process. However, experience in dealing with interactions between plan steps and the need to satisfy multiple simultaneous constraints can be transferred to the specification engineering domain.

There are a number of techniques for modifying the specification if safety violations are detected. I discuss a variety of methods here. None of these methods are currently implemented directly for the specification engineering problem. Rather, they are proposed extensions of techniques that have been shown to be useful in other artificial intelligence problem solving domains. For purposes of this dissertation the modifications are carried out by hand. The planner is then used again to determine whether the modifications have introduced any new deficiencies into the functional specification.

### Retract Previous Commitment

One approach to modification is backtracking. If a safety violation is found, some previous commitment has to be changed. Previous nondeterministic choice points must be reconsidered in case some other option would avoid the violation. When OPIE finds an inconsistency it backtracks, "undoing" the addition of an operator used in the plan that shows a constraint violation.

If a constraint violation is discovered during evaluation, the problem solver must find a non-deterministic decision that contributed to the conflict. Two primary methods of

backtracking are available [Chapman, Charniak and McDermott]. In chronological backtracking design decisions are retracted in inverse order to the order in which they were made. Chronological backtracking can also undo beneficial changes and result in combinatorial explosion.

In dependency-directed backtracking [McDermott] only decisions that directly contribute to the conflict are retracted. However, every subsequent choice that depended on that decision may need to be reconsidered. By saving the complete dependency history, it is possible to retract only the suspect decisions while preserving other, unrelated decisions made after the initial error.

#### Remove an Artifact Operator

If an operator causes a conflict in a specification, it can be removed from the set of artifact operators by the analyst.

Removing an operator can be used if one of the operators leading to the prohibited state is an artifact operator. (Environment operators are not under the control of the analyst, and cannot be removed or altered.) By removing an operator in the plan, we may make it impossible to achieve the prohibited condition.

Unfortunately, there may be an alternative operator which completes the plan or some other path to the same condition. Thus, simply removing an operator is not guaranteed to correct the problem. At best, we can try to find a single modification which “breaks” as many undesirable plans as possible.

#### Replace an Artifact Operator

Since every artifact operator was chosen to support an achievement requirement, simply deleting an operator is likely to create a deficiency.

In most cases, the operator should be replaced with an alternative which still sup-

ports the achievement requirement of the original operator, but is restricted in such a way that the prohibited condition cannot be reached.

### Restrict an Operator's Applicability

Another approach to modifying the functional specification is to provide guards for the services: restrict the class of states in which the services can be applied. This seems to be a fairly common solution to safety problems.

The implementation must include not only operators, but control knowledge for deciding when those operators should be performed. By adding control knowledge, the analyst restricts the cases in which the operator will be applied. For example, by checking to determine whether a patron's borrowing limit will be exceeded prior to checking out a book, the database satisfies both the required transition and its restriction.

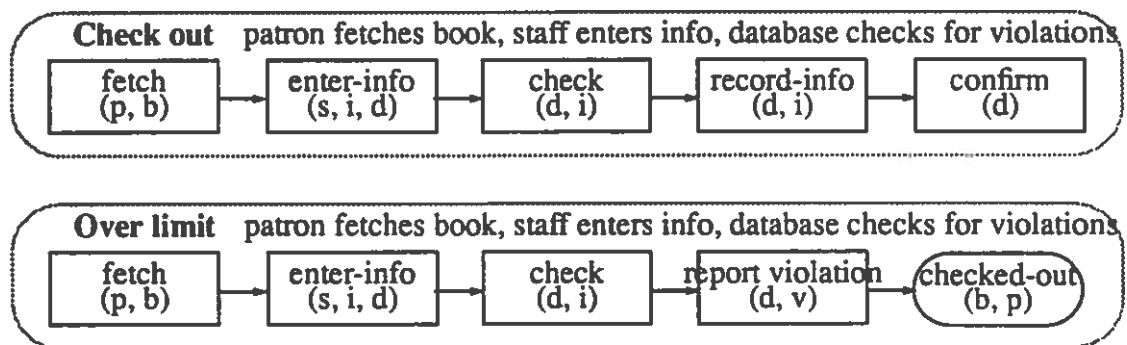


Figure 3.13 Inserting a guard to prevent a prohibited transition while still allowing a required transition. By checking to determine whether a patron's borrowing limit will be exceeded prior to checking out a book, the database satisfies both the required transition and its restriction.

One way to view the introduction of a guard is that preconditions of operators are

made more restrictive in order to disallow certain prohibited conditions. This may be viewed as a lower-grained composition problem: add additional checks and branches so that the central action is not performed under certain conditions.

Unfortunately, introducing guards cannot be implemented in the standard planning framework. The problem is that the guard does not itself contribute to the achievement of any goals. Rather, it is put in place to keep certain states from being reached. This is outside the scope of traditional artificial intelligence planning problems.

One work-around to this problem that I have employed is storing operators in both their guarded and unguarded forms. While this allows the planner to find a correct solution, in general it is not a satisfactory solution. The number of ways that a particular operator might be guarded is potentially very large, and the need to store so many versions of the same operator causes problems both for storage and for retrieval. Implementing a simple mechanism for adding guards to services seems like a better solution.

#### Disable an Environment Operator

Not all prohibited conditions can be avoided by altering artifact operators. Sometimes the prohibited condition is achieved by environment operators. Environment operators are not under the control of the analyst, and cannot be altered or removed.

The only way to prevent their use is to disable their preconditions.

It is impossible to directly modify agents (people) to change their behavior and thus change the functionality of the environment. Therefore, the analyst cannot simply remove environment operators from the composite model and assume they will not occur. Removing operators from the model of the environment does not eliminate deficiencies. Instead, it makes the model of the environment inaccurate.

Rather than remove such operators, the analyst can attempt to disable them through a process of counterplanning.



The operators can be disabled by removing operators that make their preconditions true. They can also be disabled by adding operators which make their preconditions false. Finally, they can be disabled by changing certain conditions that are relatively stable and difficult to alter relative to the lifetime of the artifact. For example, by specifying pens which are chained to the check-out counter, the analyst enables filling out check-out slips while disabling taking the pens away.

In counterplanning, as in guarding services, the goal is to establish conditions that prevent the execution of operators which would result in a prohibited state. In the case of guarded services, the analyst has control over the execution of the artifact operator being guarded. In the case of counterplanning, the analyst does not have direct control over the actions of an external actor, and therefore must use either persuasion or deterrence to elicit behavior that conforms to the client's requirements.

An environment operator can be permanently disabled by making one of its preconditions a prohibited condition. Consider the problem of keeping patrons from stealing books. The steal operator includes the precondition that the exit to the library be "free," in the sense that the thief is able to escape undetected. Thus, this steal operator can be permanently disabled by enforcing the maintenance requirement that an observer is always at the library exit. This might mean having a librarian watch everyone leaving the library, or installing an electronic detection system.

This approach has been used as a means of preventing undesirable conditions by McDermott [78] and Carbonell [81]. To prevent an event is to alter a state such that the event's preconditions are not all true, thus dis-enabling the event. [McDermott 78, p102]

Carbonell suggests a similar disable-precondition heuristic in his POLITICS system to model planning and counterplanning among human agents [Carbonell, 1981]. Along the same lines, Wilensky defines a broad model of goal conflict and planning [Wilensky, 1983], again in the realm of interacting human agents. Taking the view that the system

being specified is one active agent and the user the other, it appears that at least some of the components of Carbonell's and Wilensky's planning models can be incorporated here.

Whatever solution is chosen, it is important to then reason about the effects that decision might have on other services, such as the check-out and return operations. As an extreme example, one way to prevent stealing is to simply not have any books in the library. This will certainly have an effect on how easy it is for legitimate borrowers to obtain the materials they want.

We might consider modifying the initial state of a plan that we are trying to prevent from occurring. For instance, one library analyst suggested we could, in essence, simply remove the borrowing record from a specification to prevent the case of one user gaining access to another user's borrowing record. As even more extreme measures, we could consider removing the resource R or allow no more than one borrower B. Any or all of these modifications to the plan's initial state would keep the plan from being enabled.

#### Summary: The Planning Approach to Specification Engineering

I have implemented an automated planner to investigate the role of planning in specification engineering. Specification engineering is viewed as a deficiency driven search problem. We search in a space of functional specifications for one which satisfies the client's requirements. The search is driven by deficiencies in the current partial specification.

Deficiencies detected by the planning approach are incompleteness and unsafeness. Incompleteness occurs when a desired state transition cannot be achieved with existing services in the functional specification. Unsafeness occurs when an undesired state transition can be achieved with services in the functional specification.

Addressing incompleteness can be formulated as a planning problem. The desired state transition is presented to the planner as an initial state and a final state. The planner attempts to construct a plan which leads from the initial state to the final state. The plan

may include operators from the environment and from the catalog of possible artifact services. If a plan is found, those artifact services that contribute to the plan are included in the functional specification.

Detecting unsafeness can also be formulated as a planning problem. A state transition which represents a safety violation is presented to the planner as an initial and final state. The planner attempts to construct a plan which leads from the initial to the final state using operators from the environment and from the functional specification. If a plan is found, the functional specification fails to meet the client's requirements.

Modifying a set of services to prevent a safety violation is not accomplished directly by the planner. However, experience in artificial intelligence problem solving can be applied to the problem in a number of ways. One is to backtrack: remove some artifact service, and attempt to find an alternative service which satisfies the same positive requirements without violating the negative requirement. This can be accomplished using existing search and planning techniques, but is very inefficient.

Another approach to modifying the functional specification is to provide guards for the services: restrict the class of states in which the services can be applied. This seems to be a fairly common solution to safety problems. However, it cannot be implemented in the standard planning framework. It is a key issue for future work.

Finally, deficiencies can be addressed using counterplanning techniques to try to alter the behavior of actors in the environment rather than restricting artifact services.

One benefit of using an automated system for specification engineering is that a permanent record can be kept of the reasoning behind the set of services in the functional specification. The links between services and their roles in producing various behaviors constitute the rationale for including or excluding certain services in the specification.

## CHAPTER IV

### AVOID SEARCH USING STRUCTURED KNOWLEDGE

Traditional planners rely on search for most of their decision-making. Any problem solving method that relies on search needs to be concerned with combinatorial explosion. Combinatorial explosion refers to the exponential growth in size of the search space relative to the depth. The size of the search space can be approximated as  $O(b^d)$ , where  $b$  is the branching factor (number of children at each node), and  $d$  is the depth of the search tree (Korf, 1985).

Structuring the knowledge base helps to reduce search. Using macro-operators shortens the depth of the tree for solutions which use the macro-operators (while increasing the branching factor and therefore making performance worse on other problems). Using generalized operators allows greater control over the branching factor, moving deterministic choices higher in the tree, reducing the overall branching factor within the tree. Abstract macros can be used to divide the problem into semi-independent sub-problems. While each sub-problem requires a search space which is  $O(b^d)$ , the size of the overall search space is the sum rather than the product of these search spaces.

The use of generalized operators is an example of the notion of partial commitment. Partial commitment comes in a variety of forms. The underlying principle is that the planner is able to reduce the commitment made at a particular choice point, thus introducing information that can be used to guide other decisions, without overcommitting in a way that leads to subsequent backtracking and search. Partial commitment has been used in reducing commitment to the sequence of operators in a plan, reducing commitment to the entities which participate in an action, and, in our work, reducing commitment to the oper-

ators themselves.

The kinds of complex, real world problem solving that is required for large requirements engineering problems makes relying on weak search methods prohibitively expensive. Instead, we must provide the capability of incorporating large amounts of domain knowledge in order to avoid the high cost of blind search. The knowledge base should be organized so that the problem solver can get the maximum leverage from the information that is available.

The goal of this chapter is to describe how information is organized in OPIE and how that organization of information contributes to the performance of the planner. The knowledge base contains links between requirements and components which satisfy those requirements, links between abstractions and their specializations, and links between a whole and its parts. I describe how this organization of knowledge is useful during plan construction. I also describe extensions that would go beyond planning to reduce search during the modification phase of the specification engineering process.

### Knowledge Organization

In OPIE, each of the three kinds of schemas (object, persistence, operator) are organized into a taxonomic hierarchy. For example, `obtain` might be the parent of `make`, `buy`, `steal` and `check-out`. Each child is a specialization of its parent. I use the terms taxonomic hierarchy and generalization hierarchy interchangeably.

Furthermore, the schemas are also organized into a partonomic hierarchy. For example, one form of `check-out` may be composed of the steps `locate-book`, `bring-book-to-counter`, and `record-transaction`. Each child is a component part of its parent.

The combined taxonomic and partonomic hierarchies form an AND / OR graph. The knowledge base contains specialization and decomposition links to form these hierarchies.

### Generalization Hierarchy = Index

Operators in the knowledge base are arranged in a taxonomic hierarchy so that they can be easily retrieved when they are needed. Operators that produce the same effects are linked to a common parent. For example, a portion the operator hierarchy related to obtaining a book is shown in Figure 4.1. This organization makes it possible for OPIE to find all operators that add, for instance, the `can-use (patron, book)` condition.

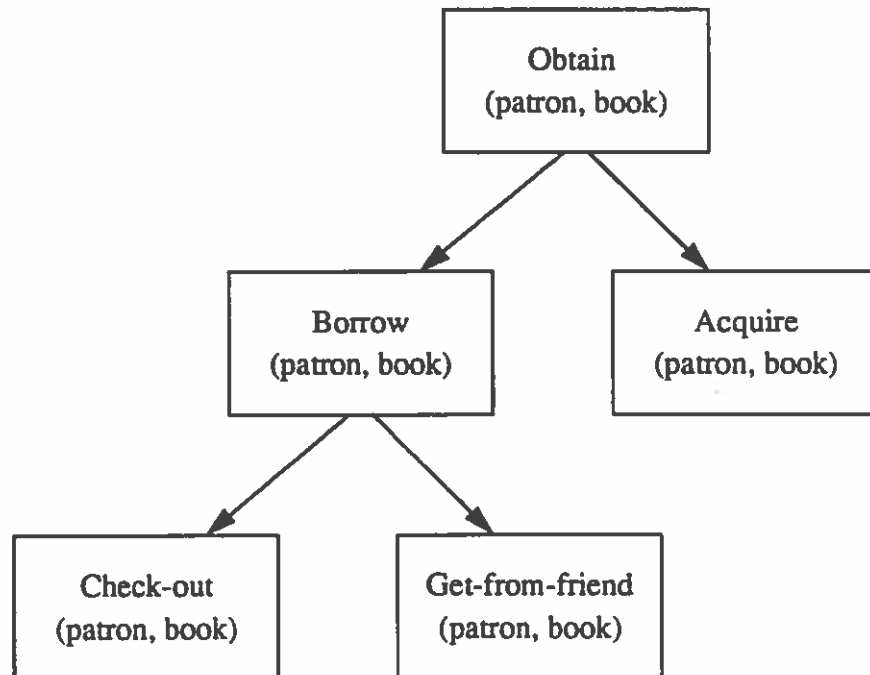


Figure 4.1 Generalization hierarchy of operators which produce `can-use (patron, book)`

The operator taxonomy is formed by a generalization process. OPIE's method of organizing operator sets is described by Anderson and Farley (1988). The organization of the knowledge base into a taxonomic hierarchy is based on the generalization of element

schemas (e.g., operators which produce common persistences). This provides an explicit representation of every choice point the analyst might encounter, which in turn provides an index to the alternatives and to selection rationale for choosing among the alternatives.

### Save and Adapt Large-grained Solution Elements

Another approach to increasing efficiency is to use larger-grained components to reduce the number of choices. Large-grained components are decomposed into smaller-grained ones by following decomposition links. For example, possible decompositions of check-out are shown in Figure 4.2.

Check out    patron fetches book and records information, staff checks and accepts

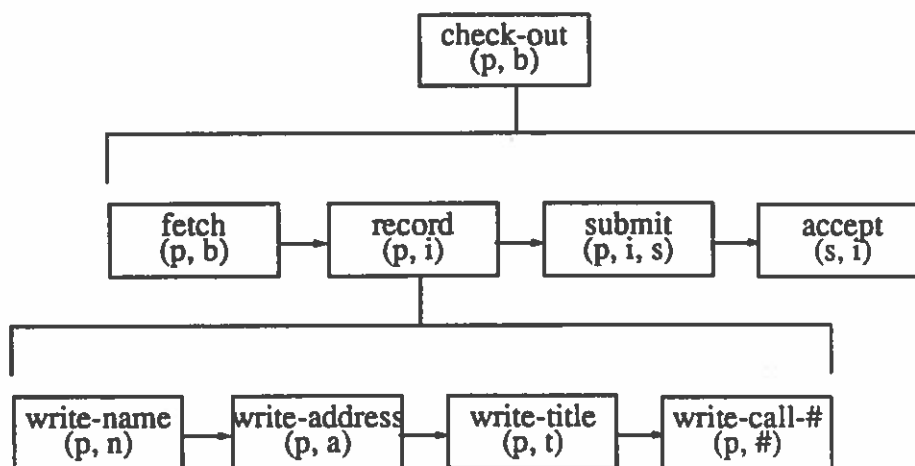


Figure 4.2 Schemas are decomposed into more detailed components in a partonomic hierarchy. p=patron, b=book, i=check-out information, s=staff

By storing large-grained solution fragments for frequently-encountered sub-problems, those solutions need not be re-derived each time. However, the fragments may

require modification to fit the current requirements.

Large-grained operators are formed by composition. Plans that successfully achieve commonly encountered goals are abstracted so that they cover as general a problem as possible and then stored as new (composite) operator schemas.

### Incremental Selection

Using search to explore all of the alternatives in a problem space is intractable for complex problems with many choice points. In search-based problem solving, one of the alternatives is selected arbitrarily and search is used to see whether that choice leads to a solution or to a contradiction. The exponential complexity of search results from a sequence of arbitrary choices, each choice requiring a separate branch in the search space.

The ideal case is to avoid the need for a problem solver to guess at any choice point. In these cases backtracking can be eliminated entirely. If a problem solver always makes the right choice, there is no need for exploring multiple branches of the search space and search is linear in the depth of the solution.

### Constraints

We distinguish primary goals from constraints, which influence the way in which the primary goals are achieved. For example, the problem “Win the war without alienating the middle class” includes a primary goal, win the war, and a constraint, avoid alienating the middle class (McDermott, 1978). Constraints cannot be executed but influence conflict resolution in deciding which method for accomplishing a primary goal should be selected.

Selecting a single option from a set of alternatives is a process of selection by elimination. Portions of the search space are gradually pruned as conflicts between constraints and plan elements are discovered. Search can be reduced if, rather than trying each of the options in turn, constraints are used to prune some of the options without explicitly con-



sidering those options.

Each choice made by the planner may introduce new constraints on the plan being composed. As decisions are made concerning the operators and objects that will be part of the plan, new constraints are introduced that affect the rest of the plan. Each new constraint may influence the choices that can be made about other plan components. The constraints are propagated through the plan. For example, the need for reference books to be continuously available might constrain the set of books that can be checked out. This in turn might exclude versions of the check-out operator that contain no way to distinguish reference books from other books.

The plan composition process is a cycle of:

- Make a choice
- Post constraints
- Look for constraint violations

#### Delay Commitment to Allow Constraints to be Introduced

Frequently, making a commitment at one choice point will introduce constraints that prune one or more options at another choice point. If the constraints are introduced early enough (i.e., before one of the pruned options is selected), some unnecessary search is avoided.

A central objective of a constraint posting approach is to make the maximum amount of information available in the form of constraints while making the fewest arbitrary decisions. More informed choices can be made if constraints are known as early as possible. If a problem solver can avoid making an arbitrary choice until constraints from other decisions are introduced, there is no need for searching alternative branches of the solution space.

This is the fundamental idea behind constraint-based planning (Stefik, 1981; Chap-

man, 1987). By delaying certain decisions until additional information is obtained from other parts of the planning process, those decisions may become sufficiently constrained that the 'choice' becomes deterministic: only one option remains. Even if the decision is not deterministic, the more it is constrained, the smaller the branching factor of the search.

The difficulty with constraint satisfaction is that the only way to introduce new constraints is to make a selection for some slot filler. This is catch-22: we want to delay making a decision until constraints can be propagated, but constraints can only be propagated if a decision is made.

In some cases, there is a solution to this dilemma. It may be possible to make a partial commitment to a set of alternatives. Partial commitment allows some of the consequences of a decision to be explored. Partial commitment to a class of options allows the problem solver to go on to other decisions without making an arbitrary choice. As other decisions are made, constraints may emerge which prune some of the alternatives without resorting to search.

Incremental selection is a form of partial commitment that both reduces the size of commitment made in a single planning decision and at the same time allows constraints to be discovered and used to prune alternatives at an earlier point in the planning process. As a result, some decisions that might lead to a non-solution branch in the search space are avoided.

The key difference between incremental selection and delayed commitment is that incremental selection not only avoids making a choice, but also allows the introduction of constraints. Any details true of all of the options under consideration can be immediately introduced, since they will be true regardless of which option is finally selected. This increases the likelihood that constraint violations will be discovered before effort is wasted in search.

### Incremental Selection Reduces Search

Incremental selection is aimed at discovering interactions at an abstract level, before the problem solver wastes effort in exploring a branch of the search space that ultimately will be rejected.

The key advantage over search-based methods is that incremental selection allows the introduction of constraints without making an arbitrary choice. Search efficiency depends on the ability to detect interactions as early as possible, before arbitrary commitments have been made. By avoiding arbitrary selections and using constraints to prune alternatives, much of the work associated with search can be avoided. For some problems, the difference between linear and exponential search lies in uncovering constraints which prune a branch before that branch is explicitly explored.

### Generalization Hierarchy Permits Incremental Selection

Organizing the knowledge base into a taxonomic hierarchy supports incremental selection, which can prune large portions of the problem space from consideration.

Generalized operators allow for partial commitment to operator selections. Using generalized plan elements increases efficiency by allowing constraints to be introduced into the plan without requiring an arbitrary choice to be made. Rather than selecting a particular operator, we use an abstract operator that generalizes all of the alternative operators that achieve the same goal. The abstract operator represents a partial commitment: a commitment to a class of operators, but not to a particular operator within that class. Guessing is avoided by proposing a general solution element that covers all of the alternatives.

By selecting an abstract operator to achieve a goal, the planner identifies the class of operators from which the actual operator will eventually be chosen. Using an abstract operator to represent a class of operators that share certain properties allows the planner to

use information about a class of operators before it has made a commitment to include a particular operator in a plan. It is thus possible for the planner to detect an interaction that would arise regardless of which operator from the set was selected.

The use of generalized components also introduces a new type of deficiency, that of ambiguity. Eventually a particular operator will have to be chosen. However, for now the planner has avoided making an uninformed decision. Rather than immediately committing to a particular method, OPIE is able to gradually prune the set of operators from which the actual operator is chosen.

Ambiguity is solved by specialization, which depends on specialization links. The generalizations can be incrementally specialized as propagated constraints reduce the number of alternatives. In the ideal case, the constraints introduced by partial commitment will be enough to determine the right choice for another decision. The outcome of that decision, in turn, may rule out some of the alternatives for the first decision. Thus, incremental selection may be able to reduce search when two or more decisions are mutually constraining.

#### Incremental Selection Example

For example, consider the case of a check-out operator that requires some identifying information to be recorded about the borrower, but does not specify what information should be recorded.

By using incremental selection, a decision need not be made immediately. Instead, a general operator that represents all of the alternatives is used at first. Now suppose attention shifts to the recall operator. In order to recall a book, the borrower must be contacted, which requires either an address or a phone number. This adds a constraint on the check-out operator, that either the address or phone number of the borrower should be recorded. Now attention shifts back to the check-out operator. Suppose one option is to use a per-

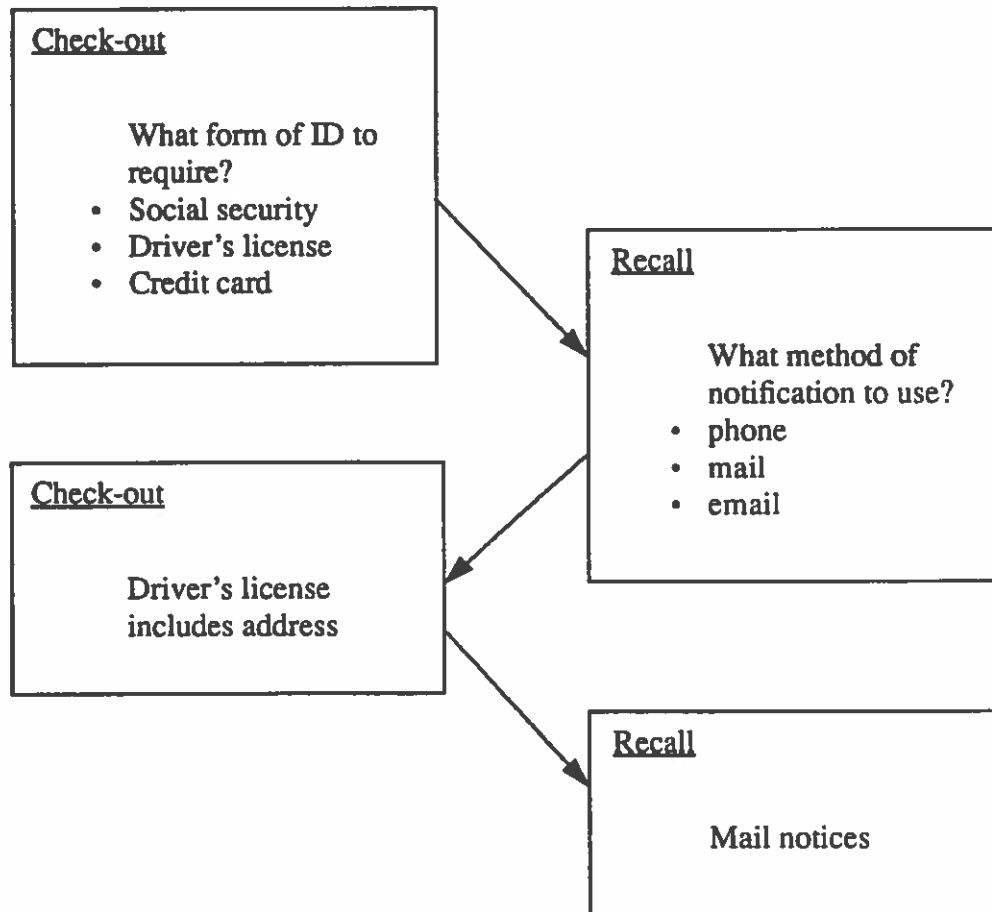


Figure 4.3 Detecting interactions efficiently: incremental selection can be used to gradually elaborate the details of a functional specification in such a way that arbitrary commitments are avoided and constraints are introduced to guide decision making.

son's driver's license as proof of identity. Furthermore, suppose driver's licenses include an address but not a phone number. Now recording the address is preferred over recording the phone number. Finally, attention goes back to the recall operator. Since it has been decided that the address will be recorded rather than the phone number, the recall operator can be specialized to using notification by mail. The complete example is shown in Figure 4.3.

This example illustrates how search is avoided by gradually refining operator

descriptions rather than generating all possible pairs of check-out and recall operators and testing to see whether they are consistent with each other.

#### Incremental Selection: Method

As discussed in Chapter 2, in order to find a plan which leads from an initial state to a goal state we must have a way of determining possible fillers for slots. This requires storing information about the possible associations among objects, persistences and operators in a knowledge base. The planner must have a library of operators which are indexed by the goals which they achieve. Schemas provide the information required for proposing candidates.

#### Access Repairs

The first step of the process is to map goals onto the operators that achieve them. This is shown in Figure 4.4. To select the operators, OPIE first matches each of the goals mentioned in an achievement requirement with an abstract persistence schema in its persistence hierarchy. Persistence schemas are linked to general operator schemas. OPIE then follows the produced-by link to find possible fillers for the slot.

For example, consider the goal of allowing a patron access to a book: `can-use (patron, book)`. Suppose we have stored the knowledge that `check-out` is an action that produces `can-use (patron, book)`. OPIE uses pattern-matching on `can-use (patron, book)` to access the corresponding persistence schema in the knowledge base. OPIE then follows the 'produced-by' link to find possible fillers for the slot.

Rather than matching every applicable operator, we match a single abstract operator which is a generalization of every operator which achieves the matching goal. For each unproduced persistence, the one "most general" operator is selected from the taxonomy.

When an operator is needed to produce a particular persistence, a single abstract

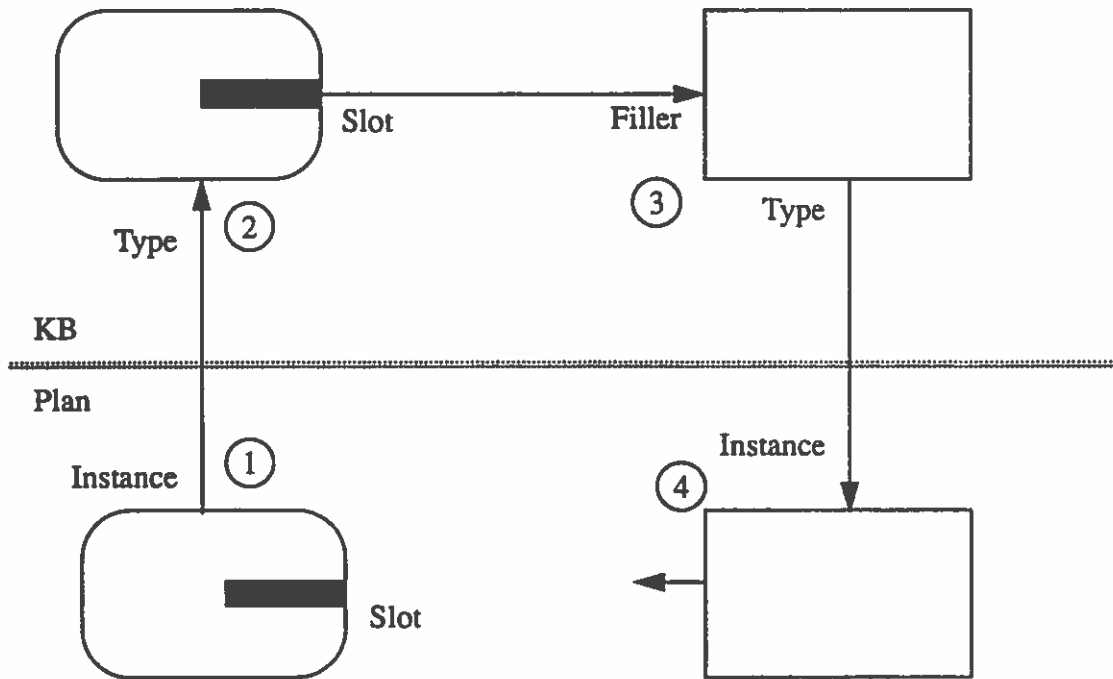


Figure 4.4 An empty slot in a plan element is filled by first tracing from the plan element to its type in the knowledge base, then finding the filler for the corresponding slot, creating a new plan element from the filler schema, and using that instance to fill the slot.

operator can be used to represent all of the alternatives. The descendants of the general operator represent alternative methods for achieving the same goal. For example, `can-use (patron, book)` is linked to a general `obtain-book` operator whose specializations might include `buy`, `steal`, and `check-out (patron, book)`.

### Specialize General Operators

We are still faced with the task of selecting a single operator from among the specializations of the abstract operator schema. OPIE simply has to select from among the alternatives. The selection process is one of filtering out alternatives that will fail, and





izations of the abstract operator is reduced.

In the ideal case, all but one of the options will eventually be eliminated by constraint propagation. Whenever only one option remains for a choice point, that option is automatically selected. Since the selection was done deterministically, there is no need for backtracking and search is avoided. To the extent that choices can be pruned in this way, unnecessary consideration of alternative branches during search is significantly reduced.

If more than one alternative remains and additional constraints are not forthcoming, all options are considered equally acceptable. In this case one of the remaining alternatives can be selected at random. However, it might turn out that the selected alternative conflicts with a constraint that is introduced later. Therefore, whenever an arbitrary choice is made it is marked as a potential backtracking point.

Each time a new specialization is selected, new constraints are introduced and propagated to the other decision points. This information may reduce the set of options for some other decision.

#### Incremental Selection: Related Work

A major theme in the evolution of planning systems has been the notion of partial commitment as a means of reducing search. The general notion is that by only making a choice when forced to do so, one avoids needless search. My work extends previous research which has shown that over-commitment can lead to unnecessary search in planning problems. Previous work has proposed partial commitment to the order of operators in a plan and to the particular objects used in a plan. OPIE uses an additional form of partial commitment: to the operators used in the plan.

NOAH (Sacerdoti, 1977) uses least commitment to temporal ordering of operators. NOAH also uses a partitioning to reduce search. MOLGEN (Stefik, 1981) uses least commitment to the objects that are used by operators, but not to the operators themselves.

Friedland's MOLGEN planner (Friedland & Iwasaki, 1985) is the most similar to OPIE. His planner selects skeletal plans which are filled in by specialization.

A few authors have described ways in which an operator taxonomy can be used to help select the right operator for a goal (Friedland & Iwasaki, 1985; Tenenberg, 1986; Alterman, 1988). Tenenberg focused on the structure of the operator taxonomy in (Tenenberg, 1986), but did not actually implement a planner that used his ideas. A process similar to incremental selection has been simulated in other automated systems using hand-coded refinement rules (Tong, 1988; Barstow, 1979).

### Abstract Plans

In addition to the use of specialization, OPIE uses decomposition as a way of reducing search. The notion is that large-grained components are introduced into the plan at first, then decomposed into their component parts.

Using large-grained components has two advantages over reasoning exclusively in terms of "atomic" components. First, since one compound component represents several choices, the total number of choices made is reduced. Since the pieces of the compound are known to fit together, this reduces the number of interactions that must be considered.

When combined with specialization, decomposition has another advantage. "Skeletal plans" are useful in problem reduction: they divide a problem into several sub-problems which can be solved (almost) independently. Any interactions are represented as constraints which further control search.

### Friedland: Refinement by Sub-planning

The most complete treatment of operator specialization is in Friedland's MOLGEN planner (Friedland & Iwasaki, 1985). Friedland's MOLGEN planner was the first to use an explicit operator taxonomy. In order to specialize operators, MOLGEN stores an operator

hierarchy in which executable operators are the leaves of the hierarchy and abstract operators are the internal nodes. The planner starts with a general plan consisting of abstract operators and refines each step of the plan until it is executable. Refinement consists of selecting one of the children of the abstract operator from the operator hierarchy. MOLGEN uses a combination of heuristics to rate the alternatives. The one with the highest total score is selected.

In Friedland's MOLGEN, the refinement process proceeds recursively until the plan consists of only executable operators. If an executable operator cannot be found for a particular step, MOLGEN makes whatever condition the step was supposed to achieve into a sub-goal. The planner is then called recursively to solve this sub-goal. If it is successful, the resultant plan is placed into the overall plan and the planner continues the refinement process.

If no such sub-plan can be found, the entire abstract plan is rejected and MOLGEN attempts to find an alternative general plan. The reason that a specific plan may not be found even though a general plan exists is that there may be interactions between the executable operators that do not appear at the abstract level. Many of the preconditions and effects of the operators have been "abstracted out" of the abstract operators, so it is not always possible to detect interactions until the operators have been specialized. Therefore, the fact that the planner finds a general plan to solve the problem does not guarantee that an executable solution actually exists.

Skeletal plans are useful not only in the domain in which they are produced, but often in other domains as well. Thus, the same general plan can be refined to solve a variety of similar problems. This is a form of leveraging control knowledge: using the same control knowledge in solving multiple problems in multiple domains.

### OPIE Combines Assignment, Specialization and Decomposition

OPIE is able to use assignment, specialization and decomposition as methods for refining a plan.

One refinement method is to assign a filler to a slot. The most important type of assignment is attributing an unproduced persistence, that is, find a producer for it. The producer may be an operator already in the plan or a new operator introduced into the plan. This is the method used in classical means-ends analysis.

An abstract plan serves as a general method of solving a problem. The plan is refined by replacing abstract operators with specific operators from the taxonomic hierarchy (Friedland & Iwasaki, 1985). For example, we might start with a general `borrow` operator and then refine it to a `check-out` operator.

Frequently, after a filler is specialized the next step is to decompose the specialization into parts. For example, the `check-out` operator might be subdivided into the steps of locating the book, bringing the book to the counter, and recording the transaction.

### Summary of the Plan Composition Process

Figure 4.6 presents the algorithm used in OPIE to compose a plan.

A plan is deficient if it contains unfilled slots and/or abstract elements. The first step is to select a deficiency to address: either a slot to fill or an element to specialize. If the deficiency selected is a missing component, the next step is to insert the component. If the deficiency selected is an abstract element, the next step is to specialize it. If the newly specialized element is composed of sub-parts, it is decomposed. Introducing or specializing an element may introduce new slots to be filled. Also, a new or specialized element may add constraints to existing slots.

The next step is to introduce and propagate constraints based on the slots associated

### Plan composition algorithm

- 0) Create a node containing the initial producer and final consumer and place it in the search queue.
- 1) Select a node from the search queue.
  - 1.a) If the search queue is empty, fail.
  - 1.b) If the node satisfies the success criteria, report success and return the node.
- 2) Refine the node:
  - 2.1) Select a plan component to refine.
  - 2.2) Generate a child node for each possible refinement.  
Refinements are assignment, specialization, and decomposition.
  - 2.3) Complete each new node by propagating constraints.
- 3) Evaluate each new node:
  - 3.1) If any constraint is violated, reject the node.
  - 3.2) Else add the node to the search queue.
- 4) Go to step 1.

Figure 4.6 Plan composition algorithm.

with the new filler or specialization. These constraints may prune non- solution components without search. Once the constraints have been introduced, the plan is evaluated for unforeseen constraint violations. If found, the plan must be revised by retracting a previous commitment and making an alternative choice.

In order to be considered finished, a plan must be complete (all slots are filled), unambiguous (all elements are completely specialized), and consistent (no constraints are violated).

### Judgment: Evaluate Options Using Stored Selection Information

The most successful method for reducing search is informed commitment: storing guidelines for making decisions based on previous problem solving experience.

It is possible to store guidelines for selecting which partial plan to refine, which deficiency within that plan to address next, and which repair for that deficiency to select. OPIE takes advantage of all of these forms of information to make planning as efficient as possible. For well defined domains, problem solving can be reduced to linear complexity by storing the right guidelines for all of the decisions encountered.

### Select a Partial Plan to Refine

Every plan cycle starts with the selection of a node to be refined. OPIE is capable of using a variety of search strategies. The simplest strategies are standard weak methods: breadth-first and depth-first search. OPIE can also use a variety of best-first search strategies, where “best” is determined using different measurements. One is the length of the plan produced so far, another is the depth of the planning process used to reach the node, another combines the number of operators in the plan with the number of remaining deficiencies. Furthermore, the human analyst is able to write arbitrary domain specific evaluation functions for ordering nodes in the search queue.

### Select a Deficiency to Address

Once a node has been chosen, the problem solver must select one of the deficiencies to address. A problem solver needs a mechanism to tell it what part of problem to work on next (Stallman & Sussman, 1977).

Strategies for choosing a deficiency to address include:

- Arbitrary selection of deficiency
- Last in, first out (LIFO) (Fikes, Hart, & Nilsson, 1972)
- Address critical deficiencies first (Sacerdoti, 1974; Knoblock, 1990)
- Address independent goals first (Christiansen, 1990)
- Make deterministic choices first (Stefik, 1981)

- Select deficiency with fewest remaining options (Anderson & Farley, 1990)
- Treat selecting a deficiency to address as a problem to be solved (Laird, Rosenbloom, & Newell, 1987)

### Evaluate Alternative Repairs

Once it has been decided which deficiency to address, the next step is to select an option to address the deficiency. For any decision, there are a variety of means for determining which alternative to choose.

- Arbitrary selection (search)
- Select an alternative using selection rules
- Rank alternatives on some scale
- Submit choices to a higher authority (ask client for preference)

The most general selection method is search: try each alternative in turn, and see whether a conflict results. Search requires the least knowledge and is the least efficient.

Another alternative is selection rules. A selection rule tells which alternative to choose under a particular set of circumstances. Selection rules are the most efficient method of selection, but require the most knowledge.

An intermediate approach is the use of heuristic selection rules, with search as a backup option. The use of heuristics allows the efficiency of selection rules whenever a rule can be found, but also gives the generality of search when no rule is available or when the rule is incorrect. This approach varies in both efficiency and knowledge requirements, depending on what rules are provided.

### Selectors: Filters and Promoters

In planning, selection rules can be associated with individual operators to determine whether that operator is appropriate under a particular set of circumstances.

When the test conditions are used to prune a candidate from the set of alternatives, they are called filters. A match filter, if true in the current state, indicates that the operator should not be considered. A non-match filter, on the other hand, indicates that the operator should not be considered if it is not true in the current state.

A non-match filter is just like a precondition, except that, if it is not true in the current state, instead of being posted as a subgoal, it simply eliminates the operator from consideration (Charniak & McDermott, 1985). Either there is no operator to achieve the precondition, or there is some other operator that would be more appropriate.

When a precondition of an operator is not achievable, there is no point in posting that precondition as a goal. If a planner cannot add a door between two rooms, it is pointless to consider an operator that requires a door that does not already exist.

The inverse of a filter is a promoter, which would make its candidate an automatic winner if its test condition was true (Laird et al., 1987). There are match and non-match selectors as well as filters.

### Taxonomy Provides Framework for Storing Selectors

The ideal case would be to have a rule for every choice point that a problem solver might encounter. However, storing a rule for every choice point in a search space is equivalent to storing the entire search space, which is not possible for complex domains. However, one can associate choice rules with the individual choice points contained in the generalization hierarchy.

One benefit of the generalization hierarchy is to allow OPIE to store the choice points and to attach the rules to the appropriate points. Generalization is a simple way of collecting the alternative methods together. By attaching locally relevant heuristics to abstract operators, domain-specific knowledge can be used to guide a general-purpose planner (Friedland & Iwasaki, 1985). The same approach is used in (Tong, 1988) in the



domain of circuit design.

### Extensions to Specification Engineering

The techniques for reducing search described in the previous sections have been implemented in OPIE and are useful for reducing search in the process of planning. My experience with OPIE suggests that these same techniques can be extended outside of the planning process itself and can be used more generally during the composition and modification of a functional specification. The extensions described in this section have not been implemented and tested specifically for specification engineering, but are fairly straight-forward extensions to the techniques that have been implemented in OPIE.

#### Adaptive Specification Using a Taxonomic Hierarchy

If an operator is used to achieve a user goal, but also achieves a prohibited condition, it may be possible to find a closely related operator in the generalization hierarchy which still achieves the goal but does not achieve the prohibited condition.

Suppose that a particular artifact operation allows a prohibited condition to be reached. One option is to replace that operator with a similar operator that achieves the same positive effects, but not the negative ones. The second operator will be in the same branch of the operator hierarchy as the first, since they share some effects. Thus, the replacement can be found by first moving up in the taxonomy, and then down again. The problem solver systematically examines nearby operators to find one which achieves the same purpose without the negative side-effect.

The basic idea for this approach has been described as adaptive planning (Alterman, 1988). The idea is to find some other operator in the taxonomy which satisfies the achievement requirement without violating the prevention requirement.

For example, consider the situation in which a person gains access to another

patron's borrowing record by querying using that person's name. The usual solution is to require some form of password before the information can be accessed. In the generalization hierarchy, the modified query (using a password) can be viewed as a sibling of the original query (without a password) which satisfies the goal of the original query without allowing the invasion of privacy.

### Compositional Specification: Using Constraints to Prune Options

Using incremental selection, it may be possible to avoid the backtracking required in the previous solution. Rather than choosing an operator and then replacing it when it turns out to be inadequate, in some cases we may be able to introduce a general operator and gradually refine it as interactions with other parts of the problem are identified.

For example, consider the interaction between the check-out and recall operators presented earlier in Figure 4.3. As long as the interaction occurs within a single plan, OPIE is able to use incremental selection to refine both operators without unnecessary search. However, if the two operators do not appear in the same plan, OPIE currently does not have any way to propagate constraints between them.

The key modification needed to extend the incremental selection approach outside of a single planning problem is to allow constraints to be propagated not only within a single planning session, but between plans as well. This extension seems like a fairly straight-forward project for future work.

### Detecting and Addressing Safety Deficiencies at an Abstract Level

Most traditional AI planners stop when they have found a single plan that achieves a goal state. In detecting deficiencies in a functional specification, however, we would like to know about all plans that lead to a prohibited condition.

Some work can be saved if critiquing a functional specification occurs at regular

intervals during the composition phase. Detecting safety violations at the abstract level has two advantages. First, since less work has been done in filling out details, less has to be undone if a deficiency is detected. Second, if a plan that leads to a prohibited condition can be disabled at an abstract level, a variety of concrete plans may be disabled at once.

If possible, we would like to know if there is one change that will disable all of the prohibited plans at the same time. Thus, it would be useful to know the common elements of all of the prohibited plans. If one precondition can be disabled or one operator can be removed to disable many or all of the undesirable scenarios, many cycles of modify and test may be avoided.

A feature of OPIE described in (Anderson & Farley, 1988) is a mechanism that generates abstract plans from executable plans by replacing the primitive operators with abstract operators.

An abstract scenario allows OPIE to find the most general plan which leads to a prohibited state. By dropping the details that are unique to one plan and preserving those aspects common to several plans, we can identify key changes which will disable a large number of similar plans. Rather than try to patch a problem shown in one particular scenario the analyst can attack the problem in abstract. OPIE can create an abstract version of the plan and can attempt to solve the more comprehensive problem.

### Macro Services

The notion of macro-operators can be extended outside of planning to the more general case of macro-decisions (Tong, 1988). One application of this idea is to group related services together into a single package. For example, the check-out and return services are highly interrelated, and typically should be selected so that they work together. The notion of macro-services is to avoid costly search when most of the plausible solutions end up being rejected due to detailed interactions.

Combining macro-decisions with generalized services provides the benefits of each, as well as dividing the search space into independent sub-spaces. For example, by selecting an abstract check-out and return macro-service, which includes a detailed description of the shared objects and persistences, the selection of particular check-out and return operators can be made independently, so long as the constraints are respected.

### Summary

Three different approaches to reducing search have been proposed in the artificial intelligence planning community. The approaches are using composite solutions to reduce the depth of the search, general solutions to control the branching factor of search, and domain specific search control knowledge to guide the choices made by the planner. All of these methods are integrated in OPIE and have significant effects on the efficiency of planning in many domains.

While the use of these methods within the planner is helpful, there is additional room for extending the methods beyond a single planning problem to assist in specification engineering decisions outside of the planner. The primary mechanism required for implementing such extensions is a means of propagating constraints between planning sessions as well as within a planning session. This extension appears to be a fairly straightforward exercise for future work.

## CHAPTER V

### RESULTS AND EVALUATION

The expected output of the specification engineering process is a functional specification of the target artifact. The inputs include a problem definition consisting of a client's requirements plus knowledge of the actions which can be performed by either the artifact or actors in the environment.

I have described the role of an automated planner in producing a functional specification given the necessary inputs. In order to evaluate the proposed theory, I have implemented a state-of-the-art planner and applied it to a number of small test problem. In this chapter I discuss the results of using the planner in solving a benchmark problem from the requirements engineering domain. The first section of this chapter presents results of running the planner on a variety of test problems which illustrate how the planner assists a human analyst.

Following the presentation of results I evaluate the planning approach to specification engineering. The evaluation of the approach is divided into five parts. Section 2 discusses expressiveness: while the use of preconditions and post-conditions to describe actions is important, it is not sufficient for expressing everything we might want to say about a functional specification. The third section discusses issues related to resolving ambiguity in the client requirements. The fourth section discusses issues related to filling in missing pieces in the client requirements. The fifth section discusses limitations of the approach for guaranteeing a complete, correct functional specification. The sixth section discusses efficiency issues.

The results of my experience indicate that planning is indeed a central component in

requirements engineering. However, extensions are needed to adapt standard planning techniques to solve requirements engineering problems. In addition, my experience provides insight into those portions of the requirements engineering process that cannot be accomplished directly by the planner. These other activities indicate the need for additional modules that must interact with the planner to complete the overall requirements engineering process.

### Empirical Study: The Library Problem

In order to evaluate the planning approach, I have applied it to a standard problem in the requirements engineering domain, the library problem.

The library problem is as close as we come to having a benchmark problem for requirements engineering. Many researchers have used the library problem to illustrate and evaluate their languages, tools and techniques for requirements engineering. A variety of different specifications of the library problem have been published. This problem was used in IWSSD4 as a benchmark problem that contributors would all use to illustrate their approaches, thus providing a common ground to help in comparing the various techniques. Furthermore, after the workshop, Jeanette Wing wrote a summary article comparing and evaluating the papers that appeared in the workshop proceedings (Wing, 1988). Other authors have continued to publish descriptions of the library problem.

Thus, the library problem is a useful problem for comparing the types of results that my automated system produces to what others believe a reasonable specification should look like. Furthermore, Wing has identified a number of issues that help to further reveal underlying assumptions and characteristics that are useful in understanding a particular approach to requirements engineering.

In the following sub-sections I discuss the results of applying the prototype system to the library problem. Figure 2.8, which presents the original problem as presented in the

call for papers of IWSSD4, is reproduced as Figure 5.1 for easy reference.

The library problem (IWSSD4, 1987):

Consider a small library database with the following transactions:

- 1- Check out a copy of a book / Return a copy of a book;
- 2- Add a copy of a book to / Remove a copy of a book from the library;
- 3- Get the list of books by a particular author or in a particular subject area;
- 4- Find out the list of books currently checked out by a particular borrower
- 5- Find out what borrower last checked out a particular copy of a book.

There are two types of users: staff users and ordinary borrowers. Transactions 1,2,4, and 5 are restricted to staff users, except that ordinary borrowers can perform transaction 4 to find out the list of books currently borrowed by themselves.

The data base must also satisfy the following constraints:

- 1- All copies in the library must be available for checkout or be checked out.
- 2- No copy of the book may be both available and checked out at the same time.
- 3- A borrower may not have more than a predefined number of books checked out at one time.

Figure 5.1 Original problem statement for the library problem.

### Results

In this section I present the results of running OPIE on several problems which illustrate its role in the specification engineering process. The first problem shows that multiple functional specifications can be found for the same set of requirements. The second problem shows how more detailed requirements can distinguish between alternative specifications. The third problem illustrates the use of the planner in finding deficiencies in a functional specification.

## Multiple Functional Specifications for the Same Requirements

In order to explore the use of OPIE in specification engineering, we developed specifications for two different library record-keeping systems, one a manual system and one an automated system. The high-level descriptions of these systems are given in Figures 5.2 and 5.3.

```

; Manual library system:
; - look-up: use card catalog
; - check-out: write on check-out slips
; - recall: staff looks for check-out slip by book
; - remind: staff looks for check-out slips by patron
; - return: find and remove check-out slips upon return
; - dual check-out slips, one indexed by patron, one by book

```

Figure 5.2 Overview of the manual library system.

The key features of the manual system that distinguish it from the automated system are the card catalog, the need to fill out paper check-out slips, and the need to use those slips for accessing information. The detailed definitions of these operators are given in Appendix A, Figure A.1.

```

; Automated library system
; - look-up: on-line catalog
; - check-out: enter patron-id and book-id
; - recall: staff queries borrowing records by book-id
; - remind: patron queries borrowing records by patron-id
; - return: delete check-out record
; - borrow records are indexed by patron-id and by book-id

```

Figure 5.3 Overview of the automated library system.



The automated system uses an on-line catalog rather than a card catalog, allows scanning of identification codes for the patron and book, and allows access to borrowing records by querying an on-line database. The detailed definitions of these operators are given in Appendix A, Figure A.2.

While there are differences between these two record-keeping systems, they both satisfy the basic requirements for a library record-keeping system, when combined with a set of environment operators. The environment operators include such things as going to the library and carrying a book from place to place. The detailed definitions of the environment operators are given in Appendix A, Figure A.3.

Figure 5.4 presents a planning problem used as input to OPIE that represents the fundamental problem of using library resources to complete some project. The problem definition includes definitions of the objects available and relationships among them.

```
(def_prob "c3"
:d "be able to complete project"
:o `( (Alice patron) (TheBook book)
      (CardCatalog catalog) (JANUS catalog)
      (Alice-id patron-id) (TheBook-id book-id) )
:i `( (available (TheBook)) (on-shelf (TheBook))
      (patron-id-of (Alice-id Alice))
      (book-id-of (TheBook-id TheBook))
      (library-owns (TheBook)) (card-catalog (CardCatalog))
      (online-catalog (JANUS)) )
:g `( (completed-project (Alice TheBook)) ) )
```

Figure 5.4 Problem definition for completing a project using a library book.

Figure 5.5 presents the solution to the problem using the manual approach. The patron goes to the library, looks up the book in the card catalog, finds the book on the shelf, checks out the book, takes the book out of the library and completes the project.

```

**** Problem: c3.1
Description: be able to complete project

**** Solution found in round 23
/----- Solution Found m7.17 (rank: 6) -----\
|   |   Init
|   - e4:{go to library} (alice)
|   - e6:{look up book} (alice thebook cardcatalog)
|   - e5:{get book} (alice thebook)
|   - e3:{check-out} (alice alice-id thebook thebook-id)
|   - e2:{Take-book-out} (alice thebook)
|   - e1:{complete project} (alice thebook)
|   |   Final
\-----/

```

Figure 5.5 Plan for completing a project requiring a library book in the case of a library with a manual record-keeping system.

Figure 5.6 presents the solution to the problem using the automated record-keeping system. The only difference visible at this level of description is that the patron looks up the book in the on-line catalog prior to going to the library. Other differences in terms of how the check-out operations are actually accomplished in the two approaches can be seen by taking a more detailed view of the plans.

One issue that needs to be addressed is that more than one patron may need the same resource. The next problem is used to ensure that it is possible for two patrons to complete their projects, where the projects both require the same resource. This problem is presented in Figure 5.7.

The solution for the manual system is presented in Figure 5.8. The solution for the automated system is similar, with the exception that looking up the book occurs before going to the library in each case.

## 2) Automated library record-keeping system

\*\*\*\* Problem: c3.1

Description: be able to complete project

\*\*\*\* Solution found in round 22

```

/----- Solution Found m6.17 (rank: 6) -----\
|   |   Init
|   - e6:{look up book on-line} (alice thebook janus)
|   - e4:{go to library} (alice)
|   - e5:{get book} (alice thebook)
|   - e3:{check-out} (alice alice-id thebook thebook-id)
|   - e2:{Take-book-out} (alice thebook)
|   - e1:{complete project} (alice thebook)
|   |   Final
\-----/

```

Figure 5.6 Plan for completing a project requiring a library book in the case of a library with an automated record-keeping system.

```

(def_prob "c4"
:d "two patrons complete projects"
:o `( (Alice patron) (Alice-id patron-id)
      (Bill patron) (Bill-id patron-id)
      (TheBook book) (TheBook-id book-id)
      (CardCatalog catalog) (JANUS catalog) )
:i `( (available (TheBook)) (on-shelf (TheBook))
      (patron-id-of (Alice-id Alice)) (patron-id-of (Bill-id Bill))
      (library-owns (TheBook)) (book-id-of (TheBook-id TheBook))
      (card-catalog (CardCatalog)) (online-catalog (JANUS)) )
:g `( (completed-project (Alice TheBook))
      (completed-project (Bill TheBook)) ) )

```

Figure 5.7 Problem definition: two patrons have projects which require the same book.

The solution given in Figure 5.8 shows one aspect of the planner which is useful in many cases of specification engineering: plans need not be totally ordered. Action e9 and e14 occur simultaneously. This is indicated by the overlapping dashes at the left of the action description.

In standard mode the planner does not actually try to discover all of the ways that actions can be performed in parallel, but allows actions to be unordered. In order to avoid potential conflicts, actions which affect the same object cannot occur simultaneously. A more sophisticated method of determining which actions can and cannot be performed simultaneously could be provided with additional effort.

```

**** Problem: c4.1
Description: two patrons complete projects

**** Solution found in round 51
/----- Solution Found m16.38 (rank: 12) -----\
|   |   Init
|   - e10:{go to library} (alice)
|   - e8:{look up book} (alice thebook cardcatalog)
|   - e6:{get book} (alice thebook)
|   - e4:{check-out} (alice alice-id thebook thebook-id)
|   - e12:{Take-book-out} (alice thebook)
|   - e2:{complete project} (alice thebook)
|   - e9:{go to library} (bill)
|   - e14:{return} (alice alice-id thebook thebook-id)
|   - e7:{look up book} (bill thebook cardcatalog)
|   - e5:{get book} (bill thebook)
|   - e3:{check-out} (bill bill-id thebook thebook-id)
|   - e11:{Take-book-out} (bill thebook)
|   - e1:{complete project} (bill thebook)
|   |   Final
\-----/

```

Figure 5.8 Plan for two patrons to complete projects requiring the same book.

### Distinguishing Between Functional Specifications

Figure 5.9 presents a problem that distinguishes between the manual and automated record-keeping systems. The problem is one of finding out that a book is currently checked out of the library. Both systems can satisfy this goal, but the automated system allows the patron to find out without actually going to the library. Thus, given a rudimen-

tary notion of relative cost, the planner can show non-functional as well as functional characteristics of plans.

```
(def_prob "f29"
  :d "know book is not available, be in office"
  :o `( (TheBook book) (TheBook-id book-id)
        (CardCatalog catalog) (OnlineCatalog catalog)
        (Bill patron) (Alice patron) (Alice-id patron-id) )
  :i `( (library-owns (TheBook))
        (card-catalog (CardCatalog))
        (online-catalog (OnlineCatalog))
        (patron-id-of (Alice-id Alice))
        (book-id-of (TheBook-id TheBook))
        (off-shelf (TheBook))
        (chkout-record (TheBook-id Alice-id))
        (in-office (Bill)) (at-terminal (Bill)) )
  :g `( (knows-book-is-out (Bill TheBook))
        (in-office (Bill)) ) )
```

Figure 5.9 Problem definition for finding out that a book is not available.

Figure 5.10 presents the solution found for the manual library. In this case, a “duration” value was given to each action, indicated by the length of the dashed line to the left of the action. The total time required for this sequence is 20 minutes.

```
**** Problem: f29.1
Description: know book is not available, be in office

**** Solution found in round 26
/----- Solution Found m10.11 (rank: 20) -----\
|   |   Init
|   ----- e2:{go to library} (bill)
|           --- e3:{look up book} (bill thebook cardcatalog)
|           ----- e1:{see book is out} (bill thebook)
|                   ----- e6:{Go to office} (bill)
|                   |   Final
\-----/
```

Figure 5.10 Plan for finding out that a book is not available in the manual library.

Figure 5.11 presents the solution for the automated record-keeping system. This solution assumes that the patron has a terminal in their office which can be used to access the library record-keeping system. In this case, the time required is 4 minutes.

```

**** Problem: f29.1
Description: know book is not available, be in office
**** Solution found in round 19
/----- Solution Found m3.14 (rank: 4) -----\
|      |      Init
|      -   e6:{log on to catalog} (bill onlinecatalog)
|      --   e2:{look up book on-line} (bill thebook onlinecatalog)
|      -   e1:{see book is out on-line} (bill thebook)
|      |      Final
\-----/

```

Figure 5.11 Plan for finding out that a book is not available in the automated library.

### Finding Deficiencies in a Functional Specification

The next problem illustrates the use of the planner in finding deficiencies in a functional specification. One of the requirements is that patrons not be allowed to check out more than some predefined number of books at a time. In order to show that this requirement can be violated, we give the planner the goal of finding a plan that results in a patron having more than the allowed number of books checked out.

The complete problem definition is shown in Appendix A, Figure A.8. Briefly, the library has a check-out limit of two books, and the goal is to show a single patron with three books checked out.

Figure 5.12 presents the plan found by OPIE for this planning problem. Because the plan indicates a deficiency in the functional specification, some change must be made.

```

**** Problem: c12.1
Description: over limit, has book
**** Solution found in round 34
/----- Solution Found m12.25 (rank: 3) -----\
|   |   Init
|   - e2:(check-out) (alice alice-id Book1 Book1-id)
|     - e3:(check-out) (alice alice-id Book2 Book2-id)
|       - e1:(check-out) (alice alice-id Book3 Book3-id)
|         |   Final
\-----/

```

Figure 5.12 Plan showing violation of borrowing limit requirement.

Figure 5.13 shows the revised check-out operator used to address this deficiency. The key change is that a record is kept of how many books a patron has checked out, and the operator is only allowed to be used when the patron is under their limit.

```

; patron goes over limit

(def_op `Restricted-check-out
 :d "Check out a book from the library"
 :o `(patron patron-id book book-id counter1 counter2 counter3)
 :- `( (available (book))
       (ready-to-check (patron book))
       (books-out1 (patron counter1))
       (books-checked1 (patron-id counter1)) )
 := `( (book-id-of (book-id book))
       (patron-id-of (patron-id patron))
       (check-out-limit (counter3))
       (follows (counter2 counter1))
       (less-than (counter1 counter3)) )
 :+ `( (can-use (patron book))
       (chkout-record (book-id patron-id))
       (books-out2 (patron counter2))
       (books-checked2 (patron-id counter2)) ) )

```

Figure 5.13 Revised operator representing a restricted form of check out. This operator cannot be applied when the patron has reached their borrowing limit.

When this operator is used to replace the original check-out operator, the safety violation no longer occurs. The planner searches in vain for a plan in which three books are checked out by the same patron at the same time.

Because the number of plans which do not violate the requirement is very large, the planner may continue indefinitely. Typically there is some time limit set on the planner. If the planner does not find a plan within that amount of time, there may still be plans which violate the safety requirement. All that the planner can tell us is that there are no plans within a bounded region of the search space.

## 2) restricted check-out

```

**** Problem: c12.1
Description: over limit, has book

* * Round 500 * * * * *
Queue: m175.35 ... (137 more)

* * TIME OUT FAILURE * * * * *
/----- Best model: m175.35 (rank: 6) -----\
| Events:
|e10: (^etyp23) (alice alice-id Book2 Book2-id zero one)
|e2: {restricted-check-out} (alice alice-id Book1 Book1-id one two
two)
|e43: (^etyp23) (alice alice-id Book2 Book2-id two zero)
|e3: {restricted-check-out} (alice alice-id Book2 Book2-id zero one
two)
|e51: (^etyp23) (alice alice-id o337 o338 one zero)
|e1: {restricted-check-out} (alice alice-id Book3 Book3-id zero one
two)
\-----/

```

Figure 5.14 Using the restricted check-out operator, the planner fails to find a plan which violates the borrowing limit within a specified time limit.

We have shown how a planner can be used in specification engineering. The remaining sections of this chapter deal with the evaluation of the planning approach to specifica-



tion engineering.

### Representation

The specification of functional specification in terms of preconditions and postconditions is a popular and generally accepted format (Wing, 1988). Of a dozen papers describing specifications of the library problem (IWSSD4, 1987), all of the papers used some sort of precondition / post-condition representation for their target specification. The common practice of describing routines in terms of their inputs and outputs is subsumed by the use of preconditions and postconditions. Thus, our output clearly fits into an acceptable format. Furthermore, given appropriate inputs, the planning approach is capable of generating specifications which are essentially the same as some of the published specifications produced by hand.

Other examples of the use of preconditions and postconditions for specification include (Reubenstein, 1990; Barstow, 1979; Kant, 1985; and Lubars & Harandi, 1989). This suggests that the output of OPIE fills the needs of designers.

### Requirements Expressed as State Transitions

Any task can be expressed in terms of a transition between pre- and post-conditions. Furthermore, any requirement to maintain or avoid certain states can also be expressed as a prohibited transition. This is shown in Figure 5.15.

### Limitation: Expressiveness

While the precondition / post-condition format is acceptable, other aspects of the representation language were inadequate. I adopted the STRIPS formalism (Fikes et al., 1972) because it was designed for planning and because it is a simple language to use. Unfortunately, this choice turned out to be less than ideal for purposes of specification.

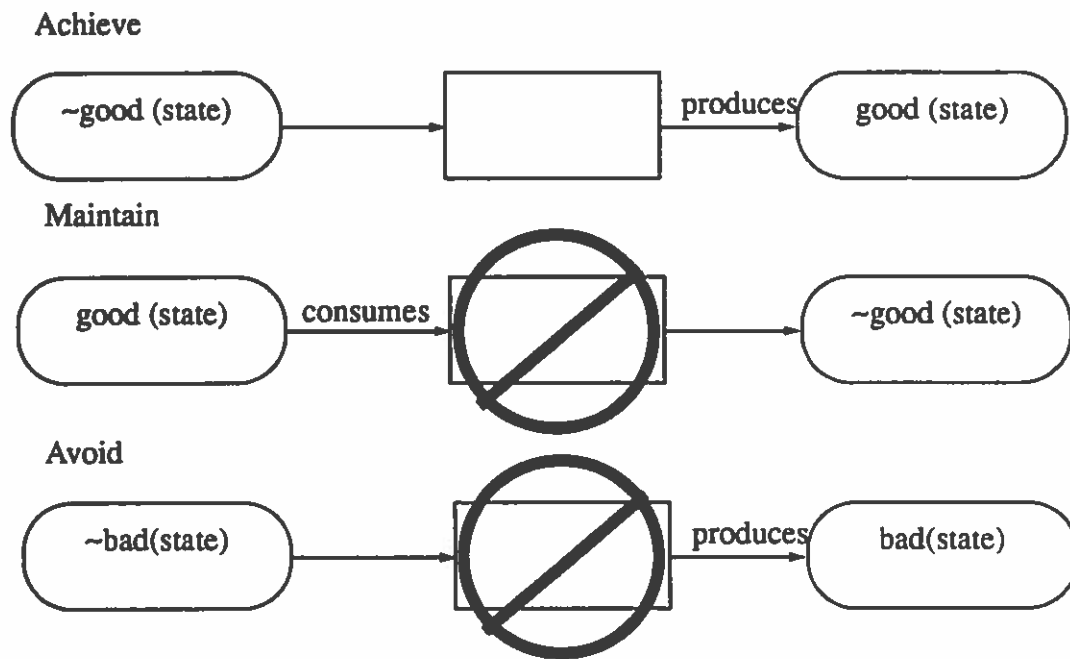


Figure 5.15 Representation of tasks as initial and final states.

The qualities demanded of a proposed representation for a model include expressiveness and analytic power. These are often in conflict because, in general, the larger the class of systems that can be described, the less is analytically decidable about them. A third important quality is naturalness of expression (Cohen, Harwood, & Jackson, 1986).

The basic STRIPS formalism is quite simple to use, but lacks some expressiveness. The STRIPS language is a restricted subset of FOL. It is possible to extend the language to increase expressiveness, but at the cost of added complexity. One trade-off is that, with a more expressive language, there is less guidance for novice users trying to use the language. Thus, while the simplicity of the language was an advantage at first, it caused problems when I tried to express more interesting problems.

The most significant lack in the STRIPS language appears to be the lack of mechanisms for manipulating sets. Universal quantification and enumeration over sets is a useful

abstraction that is used frequently in specification. In STRIPS it is assumed that sets are pre-enumerated in the initial state of the problem, so that all operators apply to individual elements rather than a set as a whole. Thus, it is not convenient to define tasks such as “find all books by an author” in the STRIPS formalism.

The use of the STRIPS language is a limitation of the current implementation, but does not appear to be an inherent problem of the planning approach. A variety of specification languages have been proposed which combine the precondition / post-condition aspects of the STRIPS formalism with mechanisms for describing operations on sets. Possible languages include VDM (Ledru, 1991), Larch (Wing & Zaremski, 1991), or a number of others (Dubois, 1989). RML (Greenspan, 1984) may be the specification language most closely aligned with the planning approach. It will be necessary to modify our system to handle these additional language constructs, but there are no obvious reasons why the extensions should not be possible.

#### Clarification: Resolving Ambiguity

In her analysis of the library problem, Wing (1988) lists a variety of issues under the heading of “ambiguity”. I classify these issues under three sub-headings:

- ambiguous system boundaries
- ambiguous intention
- ambiguous reference

#### Caveat

One caveat is that the goal of the planning approach differs from the goal of most of the other papers. Wing’s emphasis, like most of the published reports, is on moving from an informal specification to a formal one. In most cases, the specification produced was intended to be a more formal restatement of exactly the information contained in the orig-

inal.

My emphasis, on the other hand, is on going from an application-oriented description to an artifact-oriented description. The information provided by the client should be in terms of the tasks to be performed in the application domain. The specification, on the other hand, is restricted to only those functions performed by the artifact. Therefore, we expect the information content of the two descriptions to be quite different. The analyst has the freedom to make design decisions about what functionality is required to achieve the application tasks. Thus, our system does more than simply restate the initial description in a more formal way.

As a result of this difference in views, some of the issues raised are not directly relevant to the planning approach, and some of the interesting aspects of the planning approach fall outside the list of issues raised by Wing. Nevertheless, there is sufficient overlap for the exercise of discussing the issues to offer useful insights into the planning approach.

In the planning approach, the English problem statement is treated as a description of the client's objectives, rather than as an informal specification. I restated the problem formally in terms of initial and final states. This part of the process was done manually, and therefore was where most ambiguities due to natural language interpretation were resolved. However, other types of ambiguity are explicitly addressed by the planning approach.

### Ambiguous System Boundaries

The first issue Wing deals with is whether it is the library itself or a library database that is being specified. In the planning approach, the library represents the application domain and the database is the artifact to be specified. Thus, the initial description is of the library itself; that description is used to constrain the description of the database. In order

for the specification to be validated, it is important to have an integrated view of how the database interacts with the actors associated with the library.

In order to understand the tasks in which the database will be involved, we must model a significant part of the library as well. Any part of the library that interacts with the database constrains the database. At the same time, the existence of the database may force changes in the operation of the rest of the library.

In the initial stages, the boundary between the artifact and its environment is not clear. We intentionally allow the boundary to be ambiguous at first. At the top level, we have actions that involve more than one agent; the specification is not considered complete (unambiguous) until these actions have been decomposed into actions that can be performed by a single actor (Feather, 1987).

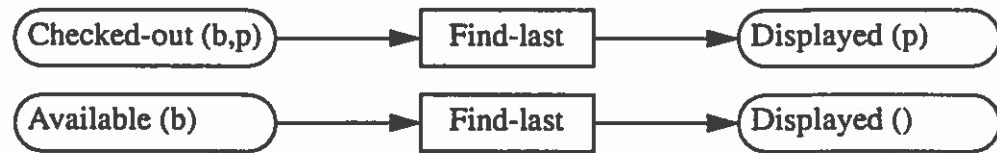
Part of the specification process, in the planning view, involves making trade-offs about whether particular operations will be performed by the artifact or by an actor (typically a human) in the environment. The trade-offs are generally decided on economic grounds: is it cheaper to try to automate this function, or to make a human responsible for this aspect of the task?

### Ambiguous Intent

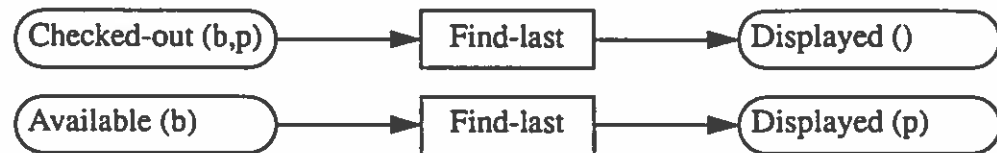
One statement in the informal specification is ambiguous with respect to what the intended action should be: what the appropriate outputs should be under different preconditions. Scenarios are useful for resolving this type of situation.

The issue is the statement that it should be possible to get a list of the patron who last checked out a book. It is unclear whether this means that, while the book is checked out, it should be possible to find out who has it (Figure 5.16a), or whether, once the book is returned, it should be possible to find out who was the last patron to have checked it out (Figure 5.16b), or both (Figure 5.16c).

(a) “last” = “current”



(b) “last” = “previous”



(c) “last” = “current or previous”

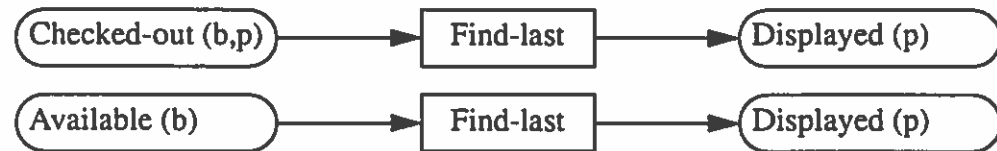


Figure 5.16 Three different interpretations of the requirement of finding out what borrower last checked out a particular copy of a book

The client requirement would have been clearer if it had been expressed in terms of initial and final states. Describing exactly what information is expected when the book is checked out and when it is not conveys the intended meaning of this statement.

This ambiguity allowed me to investigate one of the features of the planning approach that I feel is significant: the ability to produce different specifications given different requirements. I ran three tests, one using “last” = “current”, one using “last” = “previous” and one using “last” = “current or previous”. As it turns out, this distinction affects both the check-out and return scenarios, since different information must be recorded in

each case. For “last” = “current”, check-out simply records the name of the patron and return deletes it. For “last = previous”, check-out deletes the previous name and return places the name on the “last” record. In third interpretation, “last” = “current or previous”, check-out has to replace the last name with the new name; return does not affect the record.

### Ambiguous Reference

Wing points out two places where the requirements are ambiguous in terms of what a phrase refers to. The first is the distinction between a book and a particular copy of a book. The second is the restriction that books be either available or checked out.

The initial, informal specification uses the terms “book” and “copy of a book” as separate concepts. A “copy” is a physical instance of a book, “book” refers to the abstraction over all copies with the same author, title and content. In transaction 4, the informal specification uses the term “book” where “copy” seems more appropriate: “get a list of all books checked out by a patron” should be “get a list of all copies checked out by a patron”. This is an example of ambiguous communication: we can assume that the specifier had something specific in mind, but there is some question about the communication. This is the type of ambiguous reference that humans are able to resolve without conscious effort: we typically use “book” to refer to either a specific copy or to the abstraction, without much trouble.

In general, I do not claim to be able to resolve communication ambiguities. OPIE is not intended to deal with natural language issues. In this case, it does appear that using a scenario could resolve the question: a scenario in which someone lists what they have checked out could contain information about either each copy or the abstract book. A decisive scenario would be one in which the patron had two copies of the same book: presumably a list of copies would show both copies, a list of books would show just one book.

The notion of “available” seemed to need further refinement in our scenarios in order to accurately describe the period between when a book is returned and the time it is back on the shelf. We could stretch the point and consider a book available as soon as the check-out record is updated, but allowing the state of “in process” seems to better describe the situation. In fact, it seems useful to have more precise descriptions of a number of situations: after the patron has returned the book, but before the database has been updated; after a new book has been obtained, but before it has been recorded; while a patron has the book in hand, but before it has been checked out.

Our approach also revealed the notion of “exists but not owned by the library” when looking for scenarios covering what to do when a patron queries about an author not listed in the database.

### Resolving Incompleteness

The issues Wing raises under the heading of incompleteness are items missing from the initial specification that various analysts felt should have been included. Two key questions are: if the information is not in the client’s initial statement of needs, where does it come from? And how is it recognized that the information is missing? Some of the items appear to be programming knowledge that should not be expected from the client, but must be provided by the analyst’s programming knowledge. Other items require that the analyst have knowledge about the domain in order to suggest additional items that are commonly found in libraries. The analyst should ask the client whether these common items were intentionally left out or should be included. Thus, the analysts serves as a second opinion about what the requirements should include.

The incompleteness issues are: initialization of the program, missing operations, error handling, missing constraints, change of states, and non-functional requirements.



### Initialization

There is no reason for the client to be concerned with initialization of the database. However, the analyst should recognize the need and, according to Wing, include appropriate requirements in the specification.

The planning approach to specification gives explicit support for identifying the required initial state of the artifact. The preconditions of every action can be achieved in one of two ways: either by the application of another action, or by being true in the initial state. Thus, every precondition that has no producer in the combined artifact and environment action sets must be true in the initial state.

The issue of initialization has been a difficult one for my research. In the original planning approach (Anderson & Fickas, 1989) we proposed that the initial conditions of the artifact be an explicit part of the specification. This was a result of our attempts to use means-ends analysis to determine the artifact operators.

Like Wing, we originally considered the use of “create-artifact” actions which would be used to set up the database in the first place. However, in the current application-centered approach, I pay less attention to the initial state of the artifact and focus on specific tasks to be performed by the eventual users of the artifact. I have chosen to ignore the issue of initialization for now, in order to focus on the services of the specification.

### Missing Operations

There are two ways to view the operations that might be missing from a client’s initial problem description. One is that the client has left out specific tasks that need to be accomplished. The other view is that the analyst needs to fill in steps for accomplishing the client’s tasks.

The “operations” that Wing suggests adding would be seen as “transitions” in the

planning view. For example, the operation of adding a new staff person is more easily seen as a requirement that the client left out of the original problem description than as a way to achieve one of the initial transactions. The current implementation does not contain any support for the addition of new requirements. However, our view is that the analyst would have stored scenarios that could be used to point out omissions to the client as well as being used in their current role of validating against the known requirements.

I use a different notion of missing operations - operations required to satisfy the precondition of one of the operators used to accomplish a task. Using backward chaining, a planner fills in missing operations as needed to complete a plan.

There is one sense in which the two views can be merged: if, in trying to fulfill the preconditions of the steps for one task, the analyst reveals another task. For example, if one of the steps of check-out is that a staff person perform some action, then it is possible that the question of where a staff person comes from might arise.

### Error Handling

Wing incorporates checks which I would view as run-time checks: "if this situation occurs then signal an error." For example, if a person tries to check out a book when they are already at their limit, report an error rather than allowing them to check out another book. My emphasis has been on detecting errors at design time rather than at run time. If the artifact can be designed to prevent the error from occurring, do so. If not, then the types of error handling the Wing suggests is appropriate.

OPIE actively looks for the possibility of errors and recommends changes if they are found. For example, in the case of checking out too many books, OPIE generates a scenario in which a patron actually does check out too many books, and displays the plan used. The analyst can then modify the check-out scenario to include a test for number of books out, so that the check-out cannot be accomplished. This in turn results in a new ser-

vice in the specification: keeping a record of the number of books checked-out to each patron.

### Missing Constraints

The planning approach takes a knowledge-based approach to constraints: the analyst should understand the implications of whatever actions are introduced as a means of satisfying the requirements. Most of these actions have constraints in addition to the client's requirements, in the form of physical and social laws. For example, the ability to list the books checked out by a patron requires that that information be stored somewhere. The client's requirement that information be available after a book is checked out leads to constraints on the check-out operator. The information must be recorded during the check-out process in order to be available later. For example, the addition of a recall operator might require that a patron's address or phone number be recorded when a book is checked out.

### Change of State

The STRIPS approach is to only mention those persistences that are either modified by the action (produced or consumed) or are explicitly required for the action to occur (used). Wing's concern that it may not be clear in some specifications what changes and what remains the same has been explicitly addressed in the STRIPS language.

### Non-functional Behavior

Requirements often place constraints on artifact behavior in addition to desired and prohibited transitions. Other constraints include cost, performance, and reliability. In the current version of the planning approach I do not attempt to handle non-functional requirements, although I recognize the need to extend the model in the future.

### Correctness: What Can We Guarantee?

The planning process itself can be shown to be complete and correct under restricted circumstances (Chapman, 1987; Waldinger, 1977). This has been shown for single agent planning problems, given perfect knowledge and unlimited computational resources. Given a complete domain model (actions, objects and persistences) and a complete and consistent problem statement (initial and final states), a general purpose problem solver can be guaranteed to eventually return a sequence of steps that transform the initial state into the goal state, if such a sequence exists. Furthermore, if a solution is generated, it will be correct.

However, a problem solver may be theoretically complete and yet fail to find a solution to a particular problem. There are practical limitations that result from the complexity of the task and from having imperfect knowledge about a domain.

Planners have a general limitation that no plan can be proven correct in an environment with the potential for intervening events. Thus, every solution has the implicit caveat "barring unforeseen events."

The level of detail required to predict outcomes precisely is simply too fine-grained to be computationally tractable in reasonable amounts of time. The primary reason for prediction is to help determine which action to perform. Therefore, prediction does not need to be complete. However, small details can sometimes have large effects on outcomes, so even though exact prediction is not always necessary, the lack of exact prediction can sometimes be costly.

Efforts to come up with a complete and correct logic of actions which actually allow true inferences about future states are doomed to failure, since outcomes are determined by empirical states, not by necessity (Simon, 1981).

Formal methods are necessary but not sufficient for zero-fault design (Gaudel,

1991). There is reason to believe that validation is, in general, an unsolvable problem (Smith, 1985, cited in Reubenstein, 1990).

The requirements phase is inherently nonterminating, just as science is (Popper, 1959). The analyst can never ascertain that his theory is absolutely correct, although he can discover that it is not. Even to achieve the latter benefit, however, the analyst must take great care in the presentation of his theory: it must, at least, permit the inference of refutable consequences (Cohen et al., 1986).

One of an analyst's tasks is to anticipate as many sequences of external events as possible, but this is an intractable problem in the real world. One difficult aspect of this process is ensuring that the specification covers all possible sets of initial conditions from which a goal should be achieved. There is no way to guarantee correctness without considering every possible initial state. Although an automated planner can determine whether or not a condition is achievable from any particular state, it is not able to guarantee that the condition can be achieved from all possible states.

A model of the possible behavior of a complex artifact and its environment includes a virtually unlimited number of states. If we are trying to represent sequences of actions performed by users, any possible state of the artifact and its environment would be a valid starting point. There is no guarantee that the search space, or the test, is finite. OPIE cannot examine these states exhaustively.

I do not expect the planning approach to guarantee a correct specification. If there are unlimited resources available, a simpler search strategy will be just as effective. Our goal is to approach a correct specification within the limitations of practical time and resource constraints.

The solution described in this dissertation is to focus on known, high-risk safety issues. I have found that a planner is useful for finding certain kinds of constraint violations. OPIE is useful for finding known difficulties, (i.e., "here is a test case that has

caused a problem in the past: is it a problem for the current specification?"). However, we cannot guarantee that a planner will find all deficiencies in a given specification. Requirements such as "avoid condition x" can only be satisfied with respect to known operators and known initial states.

### Performance and Scalability

If a program achieves its goal eventually, it is epistemologically adequate. If a program achieves its goal in reasonable time, it is heuristically adequate (McCarthy, 1977).

In addition to achieving its goals, an automated system must be economical to use. There is little point spending more to find a problem than the problem was costing in the first place. In this section I evaluate the planning approach with respect to cost. In particular, I am interested in understanding to what extent the process can be scaled to large problems.

### Limitations: Exponential Size

An apparent problem with this approach is that the search space for finding all possible failure conditions seems exponential, if not infinite. The abstraction hierarchy reduces the complexity of the search by grouping similar failure conditions into abstract classes. The problem then becomes one of deciding how far down the hierarchy to search. One solution would be to put a time-limit on the search, and to prioritize failures in terms of 1) seriousness of the outcomes and 2) frequency. This leads to additional issues of how to obtain and store this information.

### Optimal Solutions vs. Satisficing

Finding the best solution to a problem generally takes exponential time; finding an adequate solution may be done in linear time, provided that the problem is not intrinsically

exponential and the problem solver has adequate selection criteria to make an acceptable choice at each choice point.

The space of possible specifications grows exponentially with the number of choice points. Modifying the specification involves a large number of choices. If a plan which achieves a prohibited condition is discovered, which operator(s) in that plan should be replaced? What operator(s) should be used instead to achieve the desired conditions without achieving the prohibited conditions? If these choices are not guided by heuristic selection rules, the size of the search space makes the problem intractable.

#### Issue: Planner May Generate Very Long Plan

Unfortunately, there is no guarantee that the planner will halt if it does not find a plan. Some plans can continue to grow in length indefinitely, without ever reaching a halting condition (success or failure).

One way to handle this is to take the cost of each operator into account. We place an upper bound on the cost of plans that will be considered. Plans that exceed some threshold cost can be ignored, whether considering achievement or avoidance requirements.

This opens up another opportunity for requirements violations to go undetected because the planner would not find any plans beyond that upper bound. This is known as the horizon effect in search. However, if costs are represented accurately, the plans that are overlooked will be those least likely to happen.

This limitation is less important in finding incompleteness than in finding inconsistencies because clients will often state constraints on the expense (e.g. number of steps or time required) of a plan. For example, it should not cost more to check out a book than to buy it in the store. Similarly, in most libraries the possibility of armed robbery is not a major concern.

### Scalability: Make the Right Choice or Decompose Problem

Traditional automated, general-purpose search-based problem solving methods work reasonably well in toy domains but do not appear to scale up to complex, real-world problems. The explanation is fairly straightforward. In small domains a problem solver can afford to make arbitrary choices and, if the choice turns out to be wrong, backtrack and try another option. In complex domains, the exponential cost of exploring non-solution branches makes this approach intractable.

There are two ways to scale the process of traversing a search space. The first is to store enough information such that the decision made at every choice point is the correct decision. This eliminates the need for backtracking and solution time is linear in the depth of the solution (assuming a bounded time to make each decision).

The second is to decompose the problem into independent pieces, so that the decisions made solving one section do not interact with the decisions made in another section.

When neither one of these options is available - i.e., not enough is known about the solution to always make the right choice and the problem cannot be divided into independent sub-problems, there is simply no way to guarantee a solution without investing the computation resources required by search.

One benefit from using OPIE is that all three approaches are available in the same problem solver. When there is information available, it will be used; if the problem can be sub-divided, it will be; and if weak methods are the only option, they are available as well. The program automatically uses the best option available. The planning approach maintains the safety net of general search to fall back on when knowledge-based methods fail.

The advantages of using an abstract operator hierarchy include:

- 1) Pruning is more efficient using the hierarchy because whole sets of alternatives can be pruned by a single constraint.



- 2) In addition to representing the remaining alternatives, an abstract design element introduces constraints that can restrict other design decisions.
- 3) The generalization hierarchy provides a place to store selection rules.

By itself, incremental selection improves problem solving performance to a certain extent. However, even more importantly, the abstract representations and their specializations can be stored in a taxonomic hierarchy. This makes it possible to attach context-dependent selection criteria to the abstraction. The selection information can be acquired during one problem solving session and used to avoid guessing in subsequent problem solving sessions.

#### Scenarios Must Integrate Domain and Programming Knowledge

A key prediction that this research makes is that the analyst must have integrated knowledge of the domain and the type of artifact to be built. Without the integration of the two types of knowledge, analysis becomes exponential in complexity and intractable for large problems.

By following a multi-step explanation, we reach the conclusion that analysis of large specification engineering problems requires that scenarios showing the artifact in the context of the environment must be available. It is not sufficient to supply two separate knowledge bases, one for the domain and one for the artifact. Means ends analysis is intractable for many real world problems. The general configuration task involves exponential search in the worst case (Mittal & Frayman, 1987).

Planning methods based on incremental refinement are needed for reasonably large problems. However, whereas means ends analysis can be used with arbitrary operators, refinement methods require an integrated knowledge base. Incremental refinement is also intractable in the worse case, but makes possible the storage of problem-specific control knowledge that eliminate guessing.

Thus, scenarios must include both environment and artifact actions. There are typically two ways to achieve this merger: the analyst can become an expert in the client's domain, or a prototype can be provided for the client to experience directly.

In the first case, the analyst can form an integrated model by combining knowledge about how the artifact might assist in accomplishing the tasks encountered in the domain. The analyst is able to make reasonable assumptions about how users will use the solution system based on personal experience in the domain. The analyst may acquire knowledge through observations of individuals working or by actually doing the work: the direct experience method (Marca, 1991). (Note that the direct experience approach is not available to an automated specification engineering system given current technology.)

#### Summary: Evaluation of Results

The implementation is an existence proof that the planning approach works, given appropriate input.

My conclusion regarding the effectiveness of the planning approach are for the most part positive. The planning approach produce results that would be considered to be a reasonable specification in a useful format for expressing the required services of a target artifact. The use of STRIPS operators is consistent with accepted representations in the RE community. The current representation language is less expressive for purposes of specification than other languages designed expressly for specification, but it was not designed for the same task. I expect that the planning approach can be extended by adopting a language targeted at specification problems and making appropriate extensions to the planner.

The process appears to handle a variety of issues that arise during the requirements engineering process, although my analysis has been limited to a small number of small problems.

While we can make certain theoretical guarantees about the correctness of the plan-

ning algorithms, in reality there are practical limitations that prevent us from obtaining a specification which is guaranteed to satisfy all of the client's needs. In general, the most we can guarantee is sufficiency for a prescribed set of test cases, and satisfaction within a bounded length (or cost) set of plans. Even so, the ability to automatically detect even a single constraint violation has some value.

## CHAPTER VI

### CONCLUSION

In the concluding chapter I summarize the results of this work. In addition to presenting the positive results, I discuss the large amount of work still to be done and the hard problems that still need to be addressed.

#### Research Summary

I have argued that it may be useful to view certain types of requirements engineering problems from a planning perspective. In particular, if a functional specification can be viewed as a set of services to be provided and the requirements as desired and prohibited transitions between states, then results in AI planning may be brought to bear.

To support my arguments, I have presented a prototype system which uses planning techniques for constructing and critiquing functional specifications. The planner is implemented and has been used in constructing simple functional specifications. I have implemented several test versions of my specification system; it is still an evolving, experimental system. I have applied the approach in a number of domains, including resource management (libraries), transportation (elevators, trains, intersections), and communication (email).

My research has involved proposing a particular view of the requirements engineering problem in order to take advantage of existing automated problem solving techniques, as well as proposing extensions to existing techniques to address the particular demands of requirements engineering.

My overall assessment of this work is that I have made a small step, but it appears to

be in a useful direction.

### Research Goals

The thesis proposed and investigated here is that techniques from artificial intelligence planning can be usefully applied to the problem of requirements engineering. Specifically, the planning approach addresses specification engineering for a class of artifacts known as reactive systems. Planning techniques are useful in composing functional specifications and analyzing those specifications with respect to safety requirements.

In addition to investigating the role of planning in requirements engineering, I am also interested in identifying the remaining pieces: if one assumes that planning addresses part of the requirements engineering problem, what additional processes are required to complete the picture?

My approach to investigating the thesis was to implement a state-of-the-art planner, apply it to a specific requirements engineering problem, and analyze the strengths and weaknesses of the resultant process.

### Scope of the Approach

I cannot make hard claims that this process always works or never works. Rather, I make the claim that this is a useful approach for some situations, and try to outline characteristics of the situations that it is useful for.

One distinction is between reactive and batch systems. The primary benefit of the planning approach is detecting interactions between requirements and finding a specification in which potentially conflicting requirements are resolved. In batch processing there are typically very few interactions, so the need for the planning approach may not arise. In reactive systems, users may potentially request services in any order in any combination, making reasoning about interactions much more important.

A particular type of situation in which a reactive system is used is called a composite system (Fickas, Anderson & Robinson, 1990). A composite system is one in which human, hardware and software actors engage in interactive problem solving. One example is an elevator, in which passengers contribute information about their destination and the elevator controller has to determine which direction to go and which floors to stop at based on the current set of passenger requests.

Another example is an intersection, where cars compete for a shared resource, the intersection. A traffic light controller's task is to enforce exclusive use of the intersection while maintaining some reasonable measure of fairness and progress.

### Requirements Engineering Process Model

Requirements engineering is the first stage in the development of an artifact.

The client and analyst work together toward a shared environment model. The environment model is an abstract description of the objects, relationships, and behaviors in the application environment.

The client expresses his preferences in terms of the environment model. The preferences indicate which of the states and behaviors in the environment model are desirable and which are undesirable. The requirements may also describe desired behaviors not found in the current environment model.

The analyst composes an initial functional specification of the target artifact from the client's positive requirements. The analyst constructs plans which achieve the desired states and accomplish the desired behaviors. For any step in a plan which is accomplished using an artifact service, the corresponding service becomes part of the functional specification.

The analyst then evaluates the resultant functional specification with respect to the negative requirements. If the analyst can find a plan which achieves an undesirable state

using the specified services plus environment actions, the specification is shown to be deficient. Either the functional specification must be modified, or the client's requirements must be relaxed to match the limitations of the artifact, or else the project should be abandoned.

### Specification Engineering Problem Formulation

A planning problem is expressed in terms of an initial state and a goal state. A solution is a sequence of actions which leads from the initial state to the goal state. The problem is solved by searching in the space of plans for a plan that solves the problem. In order to construct plans, a set of predefined operators is provided as input. New operators are added to the plan in response to deficiencies detected. In a planning problem, a deficiency is a goal persistence which is consumed or used but not produced in the current plan. The search space is pruned by rejecting partial plans which contain a constraint violation.

A specification engineering problem can be expressed as a set of planning problems. The client describes a set of desirable states and behaviors and a set of undesirable states and behaviors. These are represented in terms of initial and final states. A solution is a set of services to be provided by the artifact being specified. The problem is solved by searching in the space of functional specifications for a set of services that satisfies the requirements.

The view that we take is that the artifact provides a variety of services which are employed by artifact users in achieving their objectives. The set of services provided must match the needs of the user if the artifact is to be useful. That is, we are concerned with building the right system (validation) rather than building the system right (verification).

Because building the right system requires knowing whether users will be able to meet their goals, we must model and reason about actions in the application domain. It is the combination of actions by external actors and by the artifact that combine to achieve

client goals.

In order to construct functional specifications, a set of predefined services is provided as input. New services are added to the functional specification in response to deficiencies detected. A deficiency is a desirable state transition which cannot be achieved by the current functional specification.

At the same time we ensure that user goals can be achieved, we must ensure that restrictions are not violated. The search space is pruned by rejecting functional specifications which allow the violation of safety constraints.

Both of these tasks involve reasoning about sequences of actions that lead from an initial state of the world to a final state. Thus, it exactly fits the definition of a planning problem in the classical AI planning view.

### Planning Approach to Specification Engineering

In the planning approach to specification engineering, deficiencies are incompleteness and unsafeness. Incompleteness occurs when a desired state transition cannot be achieved with existing services in the functional specification. Unsafeness occurs when an undesired state transition can be achieved with services in the functional specification.

Addressing incompleteness can be formulated as a planning problem. The desired state transition is presented to the planner as an initial state and a final state. The planner attempts to construct a plan which leads from the initial state to the final state. The plan may include operators from the environment and from the catalog of possible artifact services. If a plan is found, those artifact services that contribute to the plan are included in the functional specification.

Detecting unsafeness can be formulated as a planning problem. A state transition which represents a safety violation is presented to the planner as an initial and final state. The planner attempts to construct a plan which leads from the initial to the final state using



operators from the environment and from the functional specification. If a plan is found, the functional specification fails to meet the client's requirements.

Modifying a set of services to prevent a safety violation can be accomplished in a number of ways. One is to backtrack: remove some artifact service, and attempt to find an alternative service which satisfies the same positive requirements without violating the negative requirement. This can be accomplished using existing search and planning techniques, but is very inefficient.

Another approach to modifying the functional specification is to provide guards for the services: restrict the class of states in which the services can be applied. This seems to be a fairly universal solution to safety problems. However, it cannot be implemented in the standard planning framework. It is a key issue for future work.

### Structured Knowledge

Any problem solving method that relies on search needs to be concerned with combinatorial explosion. Combinatorial explosion refers to the exponential growth in size of the search space relative to the depth. The size of the search space can be approximated as  $O(b^d)$ , where  $b$  is the branching factor (number of children at each node), and  $d$  is the depth of the search tree.

Structuring the knowledge base helps to reduce search. Using macro-operators shortens the depth of the tree for solutions which use the macro-operators (while increasing the branching factor and therefore making performance worse on other problems). Using generalized operators allows greater control over the branching factor, moving deterministic choices higher in the tree, reducing the overall branching factor within the tree. Abstract macros can be used to divide the problem into semi-independent sub-problems. While each sub-problem requires a search space which is  $O(b^d)$ , the size of the overall search space is the sum rather than the product of these search spaces.

Finally, when a problem is encountered, we can avoid backtracking in the space of plans, and simply replace the individual component that is causing the problem.

We describe how each of these techniques has been applied in the planning domain. We then describe how the techniques can be extended for reducing search in the space of functional specifications.

The notion of macro-services can be used to select a set of complementary services at the same time. For example, the check-out and return services are highly interrelated, and typically should be selected together.

Allowing generalized services to represent a variety of different specializations also gives the problem solver more control over the search space. By introducing a general check-out service, but delaying commitment to exactly which version of check-out will be provided, the problem solver increases the information content of the functional specification without making unnecessary commitments. This increases the chances that unsatisfactory alternatives will be pruned without a great deal of wasted search.

Combining macro-decisions with generalized services provides the benefits of each, as well as dividing the search space into independent sub-spaces. For example, by selecting an abstract check-out and return macro-service, which includes a detailed description of the shared objects and persistences, the selection of particular check-out and return operators can be made independently.

## Results

I have implemented a state-of-the-art planner and applied it to a specific requirements engineering problem, the library problem. The results of my experience indicate that planning is indeed a central component in requirements engineering. In addition, my experience also provides insight into those aspects of requirements engineering that cannot be accomplished by the planner, but which must interact with the planner to complete

the overall requirements engineering process.

My evaluation of the planning approach to requirements engineering is divided into five parts. The first is expressiveness: while the use of preconditions and post-conditions to describe actions is important, it is not sufficient for expressing everything we might want to say about a functional specification. The second section discusses ambiguity issues. These issues were first raised in Wing's (1988) review of a dozen papers (TWSSD4, 1987) discussing alternative approaches to the library problem. The third section is also based on Wing's review. This section discusses issues that Wing referred to as incompleteness issues. The fourth section discusses limitations of the approach for guaranteeing a complete, correct functional specification. The fifth section discusses efficiency, emphasizing the importance of structured knowledge in avoiding combinatorial explosion.

### Completing the Picture

A major weakness of the classical planning paradigm is the reliance on perfect knowledge. The approach depends on the assumption that every possible operator is described in the input set, and that the description of the initial state is complete. In reality, neither of these assumptions hold. In this section, we address the issue of incomplete knowledge within the context of our formulation of the requirements engineering problem. We propose addressing the problem by providing mechanisms for the incremental acquisition of knowledge. This acquisition is deficiency-driven: rather than trying to acquire all of the necessary information in advance, we provide methods for allowing relevant information to be supplied on demand.

The knowledge that we are concerned with includes the requirements model, the environment model, and the potential services model.

The environment model is incomplete because we do not know in advance which aspects of the environment are relevant to the current problem. As new aspects of the

problem are encountered, additional information about the domain is required.

The requirements model is incomplete because the client does not know in advance which of his preferences are relevant to the current problem. The analyst may know that certain scenarios are common, and specifically ask the client whether those scenarios are acceptable.

Finally, the potential services model may be incomplete. Not every service may be mentioned, and not every specialization of every service may be explicitly represented. If certain achievement requirements are not being met, it is useful to ask an expert in that specialty whether such a service is available. Thus, instead of trying to store complete knowledge about an area, we might instead store knowledge of who to ask about that area.

If certain safety requirements are not being met, it may be possible to provide a guard for a service that violates the safety requirement. However, it may not be possible to store in advance all possible guards for every service. Therefore, adding guards to services should be on-demand: if a situation is encountered where a safety requirement is violated, installing the appropriate guard should be done automatically or interactively.

### Benefits of Research

The two benefits of building a computer model of a process are a) we learn to better understand the process through the model, and b) the model serves as a prototype for building tools to support the process.

While OPIE is far from being a commercially viable software engineering tool, the planning approach is useful from a research perspective.

- raises questions about the requirements engineering process
- offers explanations for empirical results
- gives detailed suggestions about what information is required for requirements engineering to succeed

The model developed in this thesis is based on applying artificial planning techniques to specification engineering problems. This application has a number of benefits.

### Benefits of Scenarios

My research provides evidence which supports the thesis that scenarios play an important role in requirements engineering. This evidence adds to other work showing that scenarios play an important role in requirements engineering (Benner, 1990; Kaufman, Thebaut, & Interrante, 1989; Rosson & Carroll, 1992).

Is the planning approach to manipulating scenarios as a means of composing and validating specifications a feasible approach? The implementation serves as an existence proof of the thesis: it is indeed possible to produce a specification using the techniques which I have described and implemented. The planning technique seems useful for problems in which the artifact being specified must react to its environment.

The services provided by an artifact alter the problem space in which users do their planning. Therefore, it is necessary to model the composite system consisting of the environment and the artifact, and evaluate the composite model to determine whether client requirements are met.

Simply comparing the static composite model does not tell us whether the client requirements will be satisfied. Some method of examining the dynamic behavior of the composite system is needed. The approach advocated here is to use scenarios to predict possible consequences of fielding this artifact. Validation involves an attempt to predict the future, comparing possible scenarios to client requirements to discover deficiencies.

I believe that the basic premise proposed in the thesis is valid: planning is an essential component of the requirements engineering process. However, a number of additional pieces have to be in place before the approach is adequate for modeling the entire requirements engineering process.

### Benefits of Automation

OPIE can be viewed as a prototype tool for supporting requirements engineering using automated planning.

This work also provides some evidence regarding the use of a planner as an automated tool to perform the composition and analysis of functional specifications. What are the costs and benefits of such a tool? I am less enthusiastic about the potential for automating requirements engineering than I am for describing it. The reason is that the knowledge acquisition problem is extremely difficult.

The question can be stated as: under what conditions does the benefit of constructing a model of the artifact environment, a model of the artifact, and alternative versions of the artifact, and a task-oriented description of the requirements outweigh the cost? The answer is, whenever having the model reveals errors and deficiencies that would have gone unnoticed without the model, and would have cost more to fix than the cost of the model.

### Issues and Limitations

Requirements engineering is an ill-defined task. No single model is likely to explain all approaches to all types of requirements engineering problems. A suite of models will be required in order to cover the variety of techniques that can be applied by different analysts to different portions of a single problem, and to different kinds of problems.

There is still much that is manual about the planning approach. First, the human analyst must translate client objectives into requirements and restrictions. These must match relations that are stored in the database of the system. In addition, modification of a non-satisfying specification is currently more manual than automated. Finally, the generation of new operators requires a great deal of user intervention.

The analyst does not have any control over the planning of the user. Therefore, there

is no guarantee that the plan that the analyst selects will be the plan that the user selects. It may be necessary to check multiple paths to the same goal, or to choose one path and then inform the user of the correct steps to complete the plan (e.g., do-it yourself kits).

If you start with the right set of model components, then manipulating the model can provide useful insights into the reality being modeled. I argue that the manipulation of model components representing actions is fairly well understood for certain constrained situations. There is a large community of researchers working to improve and expand the scope of model manipulation techniques; we can benefit from that work.

However, ensuring that the model components are useful abstractions of reality is a very difficult problem. Encoding and maintaining a knowledge base of schemas from which model components are drawn is the primary bottleneck in most knowledge-based systems; ours is no exception.

Continued exploration of the relative strengths, weaknesses, and possible integration of rule-based and schema-based techniques will be an important line of research for the near future (Silverman & Mezer, 1992).

#### Future Work

- automated construction of guarded services
- language extensions: conditionals, set and member in same plan
- taking non-functional requirements into account
- knowledge acquisition mechanisms
  - interactive construction of the requirements model
  - interactive schema acquisition
  - automated acquisition of judgment

One of the central issues we are investigating involve the use of abstraction and analogy in requirements engineering. We are looking at three areas in which abstraction

and analogy appear useful. First, adaptive planning may be used to suggest modifications to the specification. Second, generalization allows extracting common features of a set of plans. Third, operators can be generated for a new domain by analogy with other domains.

### Conclusions

- planning is a significant part of requirements engineering
- decomposing requirements engineering into its information collection and model construction components should encourage progress in both areas
- automated planning techniques are useful for composing and analyzing specifications
- extensions are required to planning languages in order to make them adequate for requirements engineering problems
- the scenarios produced as a by-product of planning in specification engineering are useful for testing, documentation, maintenance and reuse
- the knowledge acquisition bottleneck limits the cost effectiveness of knowledge based systems



## APPENDIX

## DEFINITIONS

```

(def_etyp "complete project"
  :d "complete a project requiring a book"
  :o '(patron book)
  := '( (in-office (patron)) (can-use (patron book)) )
  :+ '( (completed-project (patron book)) ) )

(def_etyp "go to library"
  :d "go to a library"
  ; :at '( (duration :val 5) )
  :o '(patron)
  :] '( (in-office (patron)) )
  :+ '( (in-library (patron)) ) )

(def_etyp "get book"
  :d "get a book from the shelves"
  :o '(patron book)
  :- '( (on-shelf (book)) )
  := '( (in-library (patron))
        (available (book))
        (knows-location-for (patron book)) )
  :+ '( (ready-to-check (patron book)) ) )

(def_etyp "Take-book-out" :d "take book out of library"
  :o '(patron book)
  ; :at '( (duration :val 5) )
  := '( (can-use (patron book)) )
  :- '( (in-library (patron)) )
  :+ '( (in-office (patron)) ) )

(def_etyp "Remind-what-borrowed"
  :d "patron finds out what they have borrowed"
  :o '( "patron" patron-id book)
  := '( (knows-pid (patron patron-id)) (pid-of (patron-id patron))
        (knows-has-book (patron book patron-id)) )
  :+ '( (reminded-has-book (patron book)) ) )

(def_etyp "Guess-borrower-from-id"
  :d "patron1 finds out that borrower of book is patron2"
  :o '( "patron1" book patron2 patron-id)
  := '( (knows-has-book (patron1 book patron-id))
        (pid-of (patron-id patron2)) )
  :+ '( (knows-borrower (patron1 book patron2)) ) )

```

```

(def_etyp "look up book"
  :d "look up the location of a book"
  ; :at `( (duration :val 5) )
  :o `(patron book catalog)
  := `( (card-catalog (catalog))
        (in-library (patron))
        (library-owns (book)) )
  :+ `( (knows-location-for (patron book)) ) )

(def_etyp `Check-out
  :d "Check out a book from the library"
  :o `(patron patron-id book book-id)
  :- `( (available (book)) )
  := `( (ready-to-check (patron book))
        (bid-of (book-id book))
        (pid-of (patron-id patron)) )
  :+ `( (can-use (patron book))
        (chkout-record (book-id patron-id)) ) )

(def_etyp "Query-books-out"
  :d "staff queries for book checked out to id"
  :o `(staff patron-id book book-id)
  := `( (knows-pid (staff patron-id))
        (chkout-record (book-id patron-id))
        (bid-of (book-id book)) )
  :+ `( (knows-has-book (staff book patron-id)) ) )

(def_etyp "Query-books-out"
  :d "patron queries for book checked out to id"
  :o `(patron patron-id book book-id)
  := `( (knows-pid (patron patron-id))
        (chkout-record (book-id patron-id))
        (bid-of (book-id book)) )
  :+ `( (knows-has-book (patron book patron-id)) ) )

(def_etyp "return"
  :d "return a book"
  :o `(patron patron-id book book-id)
  :- `( (can-use (patron book))
        (chkout-record (book-id patron-id)) )
  := `( (bid-of (book-id book))
        (pid-of (patron-id patron)) )
  :+ `( (available (book))
        (on-shelf (book)) ) )

```

```

(def_etyp "look up book on-line"
  :d "look up the location for a book"
  ; :at `( (duration :val 2) )
  :o `(patron book catalog)
  := `( (online-catalog (catalog))
        (library-owns (book)) )
  :+ `( (knows-location-for (patron book)) ) )

(def_etyp 'Check-out :d "Check out a book from the library"
  :o `(patron patron-id book book-id)
  :- `( (available (book)) (ready-to-check (patron book)) )
  := `( (bid-of (book-id book)) (pid-of (patron-id patron)) )
  :+ `( (can-use (patron book)) (chkout-record (book-id patron-id)) ) )

(def_etyp "Query-books-out" :d "patron queries for book checked out
to id"
  :o `(patron patron-id book book-id)
  := `( (knows-pid (patron patron-id))
        (chkout-record (book-id patron-id))
        (bid-of (book-id book)) )
  :+ `( (knows-has-book (patron book patron-id)) ) )

(def_etyp "return"
  :d "return a book"
  :o `(patron patron-id book book-id)
  :- `( (chkout-record (book-id patron-id)) (can-use (patron book)) )
  := `( (pid-of (patron-id patron)) (bid-of (book-id book)) )
  :+ `( (available (book)) (on-shelf (book)) ) )

; patron goes over limit

(def_etyp 'Unrestricted-check-out
  :d "Check out a book from the library"
  :o `(patron patron-id book book-id counter1 counter2)
  :- `( (available (book))
        (ready-to-check (patron book))
        (books-out1 (patron counter1)) )
  := `( (bid-of (book-id book))
        (pid-of (patron-id patron))
        (follows (counter2 counter1)) )
  :+ `( (can-use (patron book))
        (chkout-record (book-id patron-id))
        (books-out2 (patron counter2)) ) )

```

```

(def_prob "c1"
:d "be able to use book"
:o `( (Alice patron) (TheBook book) (CardCatalog catalog) (JANUS
catalog)
      (Alice-id patron-id) (TheBook-id book-id) )
:i `( (library-owns (TheBook)) (card-catalog (CardCatalog))
      (online-catalog (JANUS))
      (pid-of (Alice-id Alice)) (bid-of (TheBook-id TheBook))
      (available (TheBook)) (on-shelf (TheBook)) )
:g `( (can-use (Alice TheBook)) ) )

(def_prob "c3"
:d "be able to complete project"
:o `( (Alice patron) (TheBook book) (CardCatalog catalog) (JANUS
catalog)
      (Alice-id patron-id) (TheBook-id book-id) )
:i `( (available (TheBook)) (on-shelf (TheBook))
      (pid-of (Alice-id Alice)) (bid-of (TheBook-id TheBook))
      (library-owns (TheBook)) (card-catalog (CardCatalog))
      (online-catalog (JANUS)) )
:g `( (completed-project (Alice TheBook)) ) )

(def_prob "c4"
:d "two patrons complete projects"
:o `( (Alice patron) (Alice-id patron-id)
      (Bill patron) (Bill-id patron-id)
      (TheBook book) (TheBook-id book-id)
      (CardCatalog catalog) (JANUS catalog) )
:i `( (available (TheBook)) (on-shelf (TheBook))
      (pid-of (Alice-id Alice)) (pid-of (Bill-id Bill))
      (library-owns (TheBook)) (bid-of (TheBook-id TheBook))
      (card-catalog (CardCatalog)) (online-catalog (JANUS)) )
:g `( (completed-project (Alice TheBook))
      (completed-project (Bill TheBook)) ) )

```

; Operators for looking up the location and finding (or not finding) a book

```
(def_etyp "go to library"
  :d "go to a library"
  :at `( (duration :val 6) )
  :o `(patron catalog)
  :] `( (in-office (patron)) )
  := `( (library-catalog (catalog)) )
  :+ `( (in-library (patron))
        (has-access-to (patron catalog)) ))

(def_etyp "look up book"
  :d "look up the location of a book"
  :at `( (duration :val 3) )
  :o `(patron book catalog)
  := `( (in-library (patron)) (has-access-to (patron catalog))
        (library-owns (book)) )
  :+ `( (knows-location-of (patron book)) ) )

(def_etyp "get book"
  :d "get a book from the shelves"
  :at `( (duration :val 4) )
  :o `(patron book)
  :- `( (on-shelf (book)) )
  := `( (in-library (patron))
        (available (book))
        (knows-location-of (patron book)) )
  :+ `( (ready-to-check (patron book)) ) )

(def_etyp "see book is out"
  :d "find out that a book is not available"
  :at `( (duration :val 5) )
  :o `(patron book)
  := `( (in-library (patron))
        (off-shelf (book))
        (knows-location-of (patron book)) )
  :+ `( (knows-book-is-out (patron book)) ) )

(def_etyp "Go to office" :d "go to office"
  :o `(patron)
  :at `( (duration :val 6) )
  :- `( (in-library (patron)) )
  :+ `( (in-office (patron)) ) )
```

; Operators for looking up the location and finding (or not finding) a book

```
(def_etyp "log on to catalog"
  :d "log on to electronic catalog"
  :at `( (duration :val 1) )
  :o `(patron catalog)
  := `( (online-catalog (catalog)) (at-terminal (patron)) )
  :+ `( (logged-onto (patron catalog)) ))

(def_etyp "look up book on-line"
  :d "look up the location of a book"
  :at `( (duration :val 2) )
  :o `(patron book catalog)
  := `( (at-terminal (patron)) (logged-onto (patron catalog))
        (library-owns (book)) (online-catalog (catalog)) )
  :+ `( (knows-location-for (patron book)) ))

(def_etyp "see book is out on-line"
  :d "find out on-line that a book is not available"
  :at `( (duration :val 1) )
  :o `(patron book)
  := `( (at-terminal (patron))
        (off-shelf (book))
        (knows-location-of (patron book)) )
  :+ `( (knows-book-is-out (patron book)) ))
```

```

(def_prob "f19"
:d "know location of book, be in office"
:o `( (Alice patron) (TheBook book)
      (CardCatalog catalog) (OnlineCatalog catalog)
      (Alice-id patron-id) (TheBook-id book-id) )
:i `( (library-owns (TheBook)) (library-catalog (CardCatalog))
      (online-catalog (OnlineCatalog))
      (pid-of (Alice-id Alice)) (bid-of (TheBook-id TheBook))
      (available (TheBook)) (on-shelf (TheBook)) (in-office (Alice))
      (at-terminal (Alice)) )
:g `( (knows-location-of (Alice TheBook)) (in-office (Alice)) ) )

(def_prob "f29"
:d "know book is not available, be in office"
:o `( (TheBook book) (TheBook-id book-id)
      (CardCatalog catalog) (OnlineCatalog catalog)
      (Bill patron) (Alice patron) (Alice-id patron-id) )
:i `( (library-owns (TheBook)) (library-catalog (CardCatalog))
      (online-catalog (OnlineCatalog))
      (pid-of (Alice-id Alice)) (bid-of (TheBook-id TheBook))
      (off-shelf (TheBook)) (chkout-record (TheBook-id Alice-id))
      (in-office (Bill)) (at-terminal (Bill)) )
:g `( (knows-book-is-out (Bill TheBook)) (in-office (Bill)) ) )

```

```
; check1.def
; Operators for checking out a book

(def_etyp `Write-patron-id :d "Write patron id on check-out slip"
  :o `(patron patron-id book)
  := `( (pid-of (patron-id patron)) (ready-to-check (patron book)) )
  :+ `( (pid-written (patron-id)) ) )

(def_etyp `Write-book-id :d "Write book id on check-out slip"
  :o `(patron book book-id)
  := `( (bid-of (book-id book)) (ready-to-check (patron book)) )
  :+ `( (bid-written (book-id)) ) )

(def_etyp `Check-out :d "Check out a book from the library"
  :o `(patron patron-id book book-id)
  :- `((available (book)) )
  := `( (ready-to-check (patron book))
        (pid-written (patron-id)) (bid-written (book-id))
        (bid-of (book-id book)) (pid-of (patron-id patron)) )
  :+ `( (can-use (patron book)) (chkout-record (book-id patron-id)) ) )
```



```

; check2.def
; - check-out: scan id card and barcode on book

;(def_etyp `Enter-patron-id :d "Enter patron id into database"
; :o `(patron patron-id book)
; := `( (pid-of (patron-id patron)) (ready-to-check (patron book)) )
; :+ `( (pid-entered (patron-id)) ) )

;(def_etyp `Enter-book-id :d "Enter book id into database"
; :o `(patron book book-id)
; := `( (bid-of (book-id book)) (ready-to-check (patron book)) )
; :+ `( (bid-entered (book-id)) ) )

(def_etyp `Scan-patron-id :d "Scan patron id into database"
  :o `(patron patron-id book)
  := `( (pid-of (patron-id patron)) (ready-to-check (patron book)) )
  :+ `( (pid-entered (patron-id)) ) )

(def_etyp `Scan-book-id :d "Scan book id into database"
  :o `(patron book book-id)
  := `( (bid-of (book-id book)) (ready-to-check (patron book)) )
  :+ `( (bid-entered (book-id)) ) )

(def_etyp `Check-out :d "Check out a book from the library"
  :o `(patron patron-id book book-id)
  :- `( (pid-entered (patron-id)) (bid-entered (book-id))
        (available (book)) (ready-to-check (patron book)) )
  := `( (bid-of (book-id book)) (pid-of (patron-id patron)) )
  :+ `( (can-use (patron book)) (chkout-record (book-id patron-id)) ) )

```

```

; check4.def
; patron goes over limit

(def_ptyp "can-use" :o `(patron book) )
(def_ptyp "books-out" :o `(patron counter)
  :at `( (criticality :val 3) ) )
; (stability :val 3)

(def_etyp `Unrestricted-check-out :d "Check out a book from the
library"
  :o `(patron patron-id book book-id counter1 counter2)
  :- `( (available (book)) (ready-to-check (patron book))
        (books-out1 (patron counter1)) (books-checked1 (patron-id
counter1)) )
  := `( (bid-of (book-id book)) (pid-of (patron-id patron))
        (follows (counter2 counter1)) )
  :+ `( (can-use (patron book)) (chkout-record (book-id patron-id))
        (books-out2 (patron counter2))
        (books-checked2 (patron-id counter2)) ) )

(def_etyp "Return"
  :d "return a book"
  :o `(patron patron-id book book-id counter1 counter2)
  :- `( (chkout-record (book-id patron-id)) (can-use (patron book))
        (books-out1 (patron counter1))
        (books-checked1 (patron-id counter1)) )
  := `( (pid-of (patron-id patron)) (bid-of (book-id book))
        (follows (counter1 counter2)) )
  :+ `( (available (book)) (on-shelf (book))
        (books-out2 (patron counter2))
        (books-checked2 (patron-id counter2)) ) )

```

```
; check5.def
; patron goes over limit

(def_etyp `Restricted-check-out :d "Check out a book from the
library"
  :o `(patron patron-id book book-id counter1 counter2 counter3)
  :- `( (available (book)) (ready-to-check (patron book))
        (books-out1 (patron counter1)) (books-checked1 (patron-id
counter1)) )
  := `( (bid-of (book-id book)) (pid-of (patron-id patron))
        (check-out-limit (counter3)) (follows (counter2 counter1))
        (less-than (counter1 counter3)) )
  :+ `( (can-use (patron book)) (chkout-record (book-id patron-id))
        (books-out2 (patron counter2))
        (books-checked2 (patron-id counter2)) ) )

(def_etyp "Restricted return"
  :d "return a book"
  :o `(patron patron-id book book-id counter1 counter2)
  :- `( (chkout-record (book-id patron-id)) (can-use (patron book))
        (books-out1 (patron counter1))
        (books-checked1 (patron-id counter1)) )
  := `( (pid-of (patron-id patron)) (bid-of (book-id book))
        (follows (counter1 counter2)) )
  :+ `( (available (book)) (on-shelf (book))
        (books-out2 (patron counter2))
        (books-checked2 (patron-id counter2)) ) )
```

```

(def_prob "c12"
:d "over limit, has book"
:o `( (Alice patron) (Alice-id patron-id) (TheCatalog catalog)
      (TheBook1 book) (TheBook1-id book-id)
      (TheBook2 book) (TheBook2-id book-id)
      (TheBook3 book) (TheBook3-id book-id)
      (zero counter) (one counter) (two counter) (three counter) )
:i `( (pid-of (Alice-id Alice)) (library-catalog (TheCatalog))
      (library-owns (TheBook1)) (bid-of (TheBook1-id TheBook1))
      (available (TheBook1)) (ready-to-check (Alice TheBook1))
      (library-owns (TheBook2)) (bid-of (TheBook2-id TheBook2))
      (available (TheBook2)) (ready-to-check (Alice TheBook2))
      (library-owns (TheBook3)) (bid-of (TheBook3-id TheBook3))
      (available (TheBook3)) (ready-to-check (Alice TheBook3))
      (books-out (Alice zero)) (books-checked (Alice-id zero))
      (follows (one zero)) (follows (two one)) (follows (three two))
      (less-than (zero one)) (less-than (zero two))
      (less-than (zero three)) (less-than (one two))
      (less-than (one three)) (less-than (two three))
      (check-out-limit (two)) )
:g `( (can-use (Alice TheBook1)) (can-use (Alice TheBook2))
      (can-use (Alice TheBook3)) ) )

```

## BIBLIOGRAPHY

- Allen, J., Hendler, J., & Tate, A. (Eds.). (1990). Readings in planning. San Mateo: Morgan Kaufmann.
- Alterman, R. (1988). Adaptive planning. Cognitive Science, 12, 393-421.
- Anderson, J.S. & Farley, A.M. (1988). Plan abstraction based on operator generalization. Proceedings of the Seventh National Conference on Artificial Intelligence, St. Paul, 100-104. San Mateo: Morgan Kaufmann.
- Anderson, J.S. & Farley, A.M. (1990). Incremental selection in plan composition. CIS-TR-90-11, University of Oregon.
- Anderson, J. S. & Fickas, S. (1989). A proposed perspective shift: viewing specification design as a planning problem. Proceedings of the Fifth International Workshop on Software Specification and Design, Pittsburgh, 177-184. IEEE Computer Society Press. Also appears in Partridge (Ed.), Artificial intelligence and software engineering, Ablex, 1991.
- Balzer, R., Cheatham, T.E., Jr., & Green, C. (1983, November). Software technology in the 1990's: Using a new paradigm. IEEE Computer, 16(11), 39-45. Reprinted in W.W. Agresti (Ed.), New paradigms for software development. IEEE Computer Society Press, 1986.
- Balzer, R., Goldman, N., & Wile, D. (1978, March). Informality in program specifications. IEEE Transactions on Software Engineering, 4(2), 94-102. Reprinted in C. Rich & R.C. Waters (Eds.), Readings in artificial intelligence and software engineering. Los Altos: Morgan Kaufmann, 1986.
- Barstow, D. (1979). Knowledge-based program construction. Elsevier North-Holland.
- Barstow, D. (1984, Spring). A perspective on automated programming. AI Magazine, 5-27. Reprinted in C. Rich & R.C. Waters (Eds.), Readings in artificial intelligence and software engineering. Los Altos: Morgan Kaufmann, 1986.
- Benner, K. (1990, September). Simulation in support of specification validation. Proceedings of the 5th Annual Knowledge-Based Software Assistant Conference, Liverpool, NY.
- Blum, B.I. (1992). Software engineering: A holistic view. New York: Oxford University Press.

- Boehm, B. (1984). Software life cycle factors. In C.R. Vick & C.V. Ramamoorthy (Eds.), Handbook of software engineering (pp. 494-518). New York: Van Nostrand Reinhold.
- Boehm, B. (1981). Software engineering economics. Englewood Cliffs, NJ: Prentice Hall.
- Bubenko, J.A., Jr. (1980). Information modeling in the context of system development. Proceedings of the IFIP Congress. Reprinted in P. Freeman & A.T. Wasserman (Eds.), Tutorial on software design techniques (4th ed.) (pp. 156- 172). IEEE Computer Society Press, 1983.
- Carbonell, J.G. (1981). Counterplanning: A strategy-based model of adversary planning in real-world situations. Artificial Intelligence 16, 295-329.
- Carbonell, J.G. (1983). Derivational analogy and its role in problem solving. Proceedings of the National Conference on Artificial Intelligence, 64-69.
- Chapman, D. (1987). Planning for conjunctive goals. Artificial Intelligence, 32, 333-377.
- Charniak, E. & McDermott, D.V. (1985). Introduction to artificial intelligence. Reading, MA: Addison Wesley.
- Christensen, J. (1990). A hierarchical planner that generates its own hierarchies. Proceedings of the Eighth National Conference on Artificial Intelligence, Boston, 1004-1009. Menlo Park: AAAI Press.
- Cohen, B., Harwood, W.T., & Jackson, M.I. (1986). The specification of complex systems. Reading, MA: Addison-Wesley.
- Cohen, D. (1983). Symbolic execution of the Gist specification language. Proceedings of the 8th International Conference on Artificial Intelligence, 17-20.
- Conklin, E.J. & Burgess-Yakemovic, K.C. (1991). A process-oriented approach to design rationale. Human Computer Interaction, 6(3-4).
- Curtis, B., Krasner, H., & Iscoe, N. (1988). A field study of the software design process for large systems. Communications of the ACM, 31(11), 1268-1287.
- Davis, A. M. (1990). Software requirements analysis & specification. Englewood Cliffs: Prentice Hall.
- Dean, T. & McDermott, D.V. (1987). Temporal data base management. Artificial Intelligence, 32, 1-55.
- Devanbu, P., Ballard, B.W., Brachman, R.J., & Selfridge, P.G. (1991). LaSSIE: A knowledge-based software information system. In M.R. Lowry & R.D. McCartney (Eds.), Automating software design (pp. 25-38). Menlo Park: AAAI Press.

- D'Ippolito, R.S. & Plinta, C.P. (1989). Software development using models. Proceedings of the Fifth International Workshop on Software Specification and Design, Pittsburgh, 140-142. IEEE Computer Society Press.
- Dubois, E. (1989). A logic of action for supporting goal-oriented elaborations of requirements. Proceedings of the Fifth International Workshop on Software Specification and Design, Pittsburgh, 160-168. IEEE Computer Society Press.
- Dubois E., Hagelstein J. (1987). Reasoning on formal requirements: A lift control system. Proceedings of the Fourth International Workshop on Software Specification and Design, 161-167. IEEE Computer Society Press.
- Duggins, S. (1991). A conceptual modeling approach to requirements engineering. Workshop Notes from Automating Software Design: Interactive Design, AAAI-91, Anaheim, 19-28.
- Feather, M., (1987, April). Language support for the specification and development of composite systems. ACM Transactions on Programming Languages and Systems, 9(2), 198-234.
- Fickas, S., Anderson, J., & Robinson, W. (1990, December). The Kate project: supporting specification construction. Technical Report 90-24, Computer Science Department, University of Oregon.
- Fickas, S., Collins, S., Olivier, S. (1987). Problem acquisition in software analysis: A preliminary study. Technical Report 87-04, University of Oregon.
- Fickas, S., Nagarajan, P. (1988, November). Critiquing software specifications: a knowledge based approach. IEEE Software, 37-47.
- Fikes, Richard E., Hart, P.E., & Nilsson, N.J. (1972). Learning and executing generalized robot plans. Artificial Intelligence, 3, 251-288.
- Freeman, P. & Newell, A. (1971). A model for functional reasoning in design. Proceedings of IJCAI-71, 621-640.
- Friedland, P.E. & Iwasaki, Y. (1985). The concept and implementation of skeletal plans, Technical Report KSL 85-6, Stanford University.
- Gaudel, M.-C. (1991). Advantages and limits of formal approaches for ultra-high dependability. Proceedings of the Sixth International Workshop on Software Specification and Design, Como, Italy, 237-241. IEEE Computer Society Press.
- Goedicke, M., Schumann, H., & Cramer, J. (1991). On the specification of software components. Proceedings of the Sixth International Workshop on Software Specification and Design, Como, Italy, 166-74. IEEE Computer Society Press.

- Genesereth, M.R. & Nilsson, N.J. (1987). Logical foundations of artificial intelligence. San Mateo: Morgan Kaufmann.
- Greenspan, S.J. (1984). Requirements modeling: a knowledge representation approach to software requirements definition. Technical Report CSRG-155, Computer Systems Research Group, University of Toronto, Toronto.
- Harris, D.R., Johnson, W.L., Benner, K.M., Feather, M.S. (1992). ARIES: The requirements / specification facet for KBSA. USC / ISI Technical Report.
- Hendler, J., Tate, A., & Drummond, M. (1990, Summer). AI planning: systems and techniques. AI Magazine, 11(2), 61-77.
- IWSSD4 (1987). Fourth International Workshop on Software Specification and Design, IEEE Computer Society, Order Number 769, Monterey.
- Kant, E. (1985, November). Understanding and automating algorithm design. IEEE Transactions on Software Engineering, 11, 1361-1374.
- Kaufman, L. D., Thebaut, S. M., & Interrante, M. F. (1989, September). System modeling for scenario-based requirements engineering. SERC-TR-33-F.
- Knoblock, C. (1990). Learning abstraction hierarchies for problem solving. Proceedings of the Eighth National Conference on Artificial Intelligence, Boston, 923-928. Menlo Park: AAAI Press.
- [Korf, R. (1985). Planning as search: a quantitative approach. Artificial Intelligence, 33: 65-88.
- Kramer, J., Magee, J., & Sloman, M. (1989). Configuration support for systems description, construction and evaluation. Fifth International Workshop on Software Specification and Design, Pittsburgh, 28-33.
- Laird, J.,E., Rosenbloom, P.S., & Newell, A. (1987). Soar: An architecture for general intelligence. Artificial Intelligence, 33(1), 1-64.
- Ledru, Y. (1991). Developing reactive systems in a VDM framework. Sixth International Workshop on Software Specification and Design, Como, Italy, 130-139.
- Lubars, M.D. & Harandi, M.T. (1989). Addressing software reuse through knowledge-based design. In Biggerstaff & Perlis, (Eds.), Software reusability Vol II. ACM Press.
- Lubars, M.D., Potts, C., & Richter, C. (1993). A review of the state of the practice in requirements modeling. Proceedings of the IEEE International Symposium on Requirements Engineering, IEEE Computer Society Press.



- Maarek, Y.S. & Berry, D.M. (1989). The use of lexical affinities in requirements abstraction. Fifth International Workshop on Software Specification and Design, Pittsburgh, 196-202.
- Marca, D.A. (1991). Specifying groupware requirements from direct experience. Sixth International Workshop on Software Specification and Design, Como, Italy, 224-232.
- McCarthy, J. (1977). Epistemological problems of artificial intelligence. Proceedings of IJCAI-77, Cambridge, MA. Reprinted in R.J. Brachman & H.J. Levesque (Eds.), Readings in Knowledge Representation. Los Altos: Morgan Kaufmann, 1985.
- McDermott, D. (1978). Planning and acting. Cognitive Science, 2, 71-109.
- McDermott, D. (1982). A temporal logic for reasoning about processes and plans. Cognitive Science, 6(2), 101-155.
- Mittal, S. & Frayman, F. (1987). Making partial choices in constraint reasoning problems, Proceedings of AAAI-87, Seattle, 631-636.
- Mostow, J. (1985, Spring). Toward better models of the design process. AI Magazine, 6(1), 44-57.
- Newell, A. (1969). Heuristic programming: Ill structured problems. In J. Aronofsky (Ed.), Progress in operations research (pp. 360-414). New York: Wiley & Sons.
- Nilsson, N.J. (1980). Principles of artificial intelligence. Los Altos: Morgan Kaufmann.
- Popper, K., (1959). The logic of scientific discovery. London: Routledge and Kegan Paul.
- Reubenstein, H.B. (1990). Automated acquisition of evolving informal descriptions. Ph.D. Thesis, MIT, Boston.
- Reubenstein, H.B. & Waters, R.C. (1991). The Requirements Apprentice: automated assistance for requirements acquisition, IEEE Transactions on Software Engineering.
- Rich, C. & Waters, R.C. (Eds.). (1986). Readings in artificial intelligence and software engineering. Los Altos: Morgan Kaufmann.
- Roman, G.-C. (1985, April). A taxonomy of current issues in requirements engineering. Computer, 14-22. Reprinted in R.H. Thayer & M. Dorfman (Eds.), System and software requirements engineering. IEEE Computer Society Press, 1990.
- Rosson, M. B. & Carroll, J. M. (1992, March). Extending the task-artifact framework: scenario-based design of smalltalk applications, Research Report RC 17852 (#78516), IBM Research Division, T.J. Watson Research Center.

- Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W. (1991). Object-oriented modeling and design. Englewood Cliffs, NJ: Prentice-Hall.
- Sacerdoti, E. (1974). Planning in a hierarchy of abstraction spaces. Artificial Intelligence, 5, 115-135.
- Sacerdoti, E. (1977). A structure for plans and behavior. New York: American Elsevier.
- Silverman, B.G. & Mezher, T.M. (1992, Spring). Expert critics in engineering design: Lessons learned and research needs. AI Magazine, 13(1), 45-62.
- Simon, H. (1981). The sciences of the artificial (2nd ed.). Cambridge, MA: MIT Press.
- Smith, B. (1985). The limits of correctness. ACM SIGCAS Computers and Society, 14/15(4/1,2,3), 18-26.
- Soloway, E. & Erlich, K. (1984). Empirical studies of programming knowledge. IEEE Transactions on Software Engineering 10(5), 595-609. Reprinted in C. Rich & R.C. Waters (Eds.), Readings in artificial intelligence and software engineering. Los Altos: Morgan Kaufmann, 1986.
- Stallman, R. & Sussman, G. (1977). Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. Artificial Intelligence 9, 135-196.
- Stefik, M.J. (1981). Planning with constraints (MOLGEN: Part 1). Artificial Intelligence, 16, 111-140.
- Steier, D. (1989). Automating algorithm design within a general architecture for intelligence. Ph.D. Dissertation, Carnegie Mellon University.
- Tenenberg, J. (1986). Planning with abstraction. Proceedings of the Fifth National Conference on Artificial Intelligence, Philadelphia, PA, 76-80.
- Thayer, R.H. & Dorfman, M. (Eds.). (1990). System and software requirements engineering. IEEE Computer Society Press.
- Tong, C. (1988). Knowledge-based circuit design. Ph.D. dissertation, Stanford University, LCSR-TR-108, Rutgers University.
- Waldinger, R. (1977). Achieving several goals simultaneously. In E. Elcock & D. Michie (Eds.), Machine Intelligence 8 (pp. 94-136). Ellis Horwood.
- Wexelblat, A., (1987). Report on scenario technology. MCC Technical Report Number STP-139-87.
- Wile, D. S., (1983). Program developments: formal explanations of implementations. Communications of the Association for Computing Machinery, 26(11).

- Wilensky, R. (1983). Planning and understanding. Reading, MA: Addison-Wesley.
- Wilkins, D. (1988). Practical planning: Extending the classical AI paradigm. San Mateo: Morgan Kaufmann.
- Wing, J. (1988, July). A study of 12 specifications of the library problem. IEEE Software, 66-76.
- Wing, J.M. & Zaremski, A.M. (1991). A formal specification of a visual language editor. Sixth International Workshop on Software Specification and Design, Como, Italy, 120-129.
- Yeh, R.T., Zave, P., Conn, A.P., & Cole, G.E., Jr. (1984). Software requirements: new directions and perspectives. In C.R. Vick & C.V. Ramamoorthy (Eds.), Handbook of software engineering (pp. 519-543). New York: Van Nostrand Reinhold.

