

AUTOMATED SUPPORT FOR REQUIREMENTS TRANSFORMATION  
IN SOFTWARE ENGINEERING

by

BRIAN LEE DURNEY

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

June 1994

"Automated Support for Requirements Transformation in Software Engineering," a dissertation prepared by Brian L. Durney in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science.

This dissertation has been approved and accepted by:



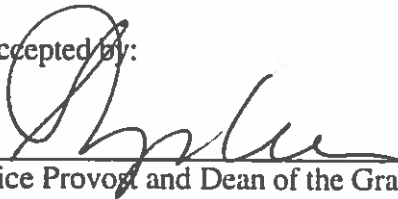
Dr. Stephen Fickas, Chair of the Examining Committee

5/9/94

Date

Committee in charge: Dr. Stephen Fickas, Chair  
Dr. Arthur Farley  
Dr. Virginia Lo  
Dr. Edward Weeks


Accepted by:



Vice Provost and Dean of the Graduate School

An Abstract of the Dissertation of  
Brian Lee Durney for the degree of Doctor of Philosophy  
in the Department of Computer and Information Science  
to be taken June 1994

Title: AUTOMATED SUPPORT FOR REQUIREMENTS TRANSFORMATION  
IN SOFTWARE ENGINEERING

Approved:   
Dr. Stephen F. Fickas

A requirement that is difficult or impossible to satisfy can lead to one of two extremes: (1) an unsatisfied, impractical requirement that is too strong, or (2) a requirement that has been unnecessarily weakened or abandoned. Traditional software engineering methods are not well suited to addressing problems caused by requirements that are too weak or too strong because such methods focus on changing specifications, not requirements.

The premise of this work is that changing requirements is an important alternative to changing specifications. My thesis is that knowledge-based analysis of requirements and environmental constraints supports requirements change by allowing a program to find the strongest requirement satisfied by a specification.

This dissertation makes two contributions. First, it defines a relation called IS-STRONGER-THAN that allows a program to compare the strength of alternative requirements. A program called GIRAFFE uses a set of transformations based on the IS-STRONGER-THAN relation to incrementally strengthen and weaken requirements. Second, the dissertation describes a method for finding general scenarios. GIRAFFE uses

general scenarios to determine when requirements changes are appropriate. GIRAFFE's method for finding scenarios allows it to use partially-specified initial states and find scenarios that include multiple paths and more general object types.

GIRAFFE uses the IS-STRONGER-THAN relation and general scenarios to find the strongest requirement satisfied by a specification. The program thus helps an analyst change requirements and so avoid the problems of unsatisfied, impractical requirements and unnecessarily weak requirements.

## CURRICULUM VITA

NAME OF AUTHOR: Brian Lee Durney

PLACE OF BIRTH: Seattle, Washington

DATE OF BIRTH: August 21, 1959

### GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon  
Stanford University  
University of Utah

### DEGREES AWARDED:

Doctor of Philosophy in Computer Science, 1994, University of Oregon  
Master of Science in Computer Science, 1985, Stanford University  
Bachelor of Science in Computer Science, 1984, University of Utah

### AREAS OF SPECIAL INTEREST:

Artificial Intelligence  
Software Engineering

### PROFESSIONAL EXPERIENCE:

Teaching and Research Assistant, Department of Computer and Information  
Science, University of Oregon, Eugene, 1987-93.

Member of Technical Staff, AT&T Bell Laboratories, Naperville, Illinois, 1984-87.

### AWARDS AND HONORS:

National Merit Scholarship, 1977

Magna cum laud, 1984

**PUBLICATIONS:**

Anderson, J. S., & Durney, B. (1993). Using scenarios in deficiency-driven requirements engineering. In *Proceedings of the IEEE International Symposium On Requirements Engineering* (pp. 134-141). Los Alamitos, CA: IEEE Computer Society.

## ACKNOWLEDGEMENTS

I thank Steve Fickas, my advisor, for his help. I thank John Anderson, Anne Dardenne, Rob Helm and Bill Robinson for their comments, discussion, suggestions, and other help. I also thank Art Farley, Ginnie Lo, and Ed Weeks for their comments.

The research presented in this dissertation was supported in part by a grant from the National Science Foundation to Dr. Stephen Fickas at the University of Oregon.

I thank Kristy, Jessica and Alexander for their patience and help, and for a lot of fun over the past seven years.

## TABLE OF CONTENTS

| Chapter  | Page |
|--|------|
| I. INTRODUCTION .....                          | 1    |
| Requirements Transformation .....              | 4    |
| Why Transform Requirements?.....               | 12   |
| Thesis Statement .....                         | 14   |
| Assumptions.....                               | 14   |
| Contributions.....                             | 15   |
| Overview.....                                  | 16   |
| II. GIRAFFE'S MODEL AND PROCESS.....           | 18   |
| Introduction.....                              | 18   |
| Representation.....                            | 18   |
| The Process.....                               | 32   |
| Related Work.....                              | 34   |
| Summary.....                                   | 49   |
| III. REQUIREMENTS RELATIONS.....               | 50   |
| Introduction.....                              | 50   |
| Definition of a Requirements Relation.....     | 50   |
| Other Requirements Relations.....              | 67   |
| Related Work.....                              | 74   |
| Summary.....                                   | 85   |
| IV. GIRAFFE'S KNOWLEDGE BASE.....              | 87   |
| Introduction.....                              | 87   |
| Applicability of Transformations.....          | 88   |
| Effects of Transformations .....               | 93   |
| Rating Functions.....                          | 100  |
| Summary.....                                   | 107  |
| V. FINDING SCENARIOS.....                      | 108  |
| Introduction.....                              | 108  |
| General Scenarios.....                         | 108  |
| Using Planning Methods to Find Scenarios ..... | 109  |
| Other Methods for Finding Scenarios .....      | 120  |
| Summary.....                                   | 131  |



| Chapter                                | Page |
|--|------|
| VI. EVALUATION.....                    | 133  |
| Introduction.....                      | 133  |
| Evaluation of the Implementation ..... | 133  |
| Generality of the Implementation ..... | 151  |
| Evaluation of the Method .....         | 157  |
| Summary.....                           | 159  |
| VII. CONCLUSION .....                  | 161  |
| Introduction.....                      | 161  |
| Contributions.....                     | 161  |
| Limitations and Future Work.....       | 162  |
| Conclusion.....                        | 165  |
| APPENDIX                               |      |
| A. GLOSSARY .....                      | 167  |
| B. DOMAIN MODEL.....                   | 171  |
| C. EXAMPLES .....                      | 190  |
| REFERENCES.....                        | 211  |

## LIST OF FIGURES

| Figure   | Page |
|--|------|
| 1. A Hypothetical Conversation Between an Analyst and a Client .....                           | 6    |
| 2. Informal Statements of Two Requirements<br>for an On-Line Registration System.....          | 7    |
| 3. Transformation of Satisfied and Target Requirements.....                                    | 8    |
| 4. Parts of a Requirements Lattice .....   | 10   |
| 5. A Transition that Represents an Achievement Requirement.....                                | 19   |
| 6. Summary of GIRAFFE's Representation of Requirements.....                                    | 21   |
| 7. A Transition that Represents a Safety Requirement .....                                     | 22   |
| 8. Two Qualifications for a Requirement .....  | 24   |
| 9. An Operator that Represents a Capability of an Artifact .....                               | 28   |
| 10. Examples of the Three Capability Sets Used by GIRAFFE.....                                 | 29   |
| 11. A Scenario.....  | 31   |
| 12. The Process that GIRAFFE Uses to Transform Requirements.....                               | 32   |
| 13. Two Requirements Stated as Prohibited or Desired Transitions.....                          | 37   |
| 14. Comparison of the Representations that OPIE, GIRAFFE<br>and ISAT Use for Requirements..... | 44   |
| 15. Definition of SET-STRONGER-THAN Relation Between<br>Sets of Requirements.....              | 51   |
| 16. Definition of IS-STRONGER-THAN for<br>Achievement Requirements.....                        | 51   |
| 17. Definition of IS-STRONGER-THAN for<br>Safety Requirements .....                            | 52   |
| 18. Definition of H-IS-STRONGER-THAN.....  | 53   |

| Figure  | Page |
|---|------|
| 19. Sets of Scenarios .....   | 58   |
| 20. Comparison of Two Transitions Based on<br>Generality of Object Types .....  | 60   |
| 21. Comparison of Two Transitions Based on<br>AND/OR expressions .....          | 63   |
| 22. Some Classes of Conditions .....  | 63   |
| 23. A Case where the Heuristic Rule H1 Is Misleading .....                      | 65   |
| 24. A Support Requirement Derived from<br>an Achievement Requirement .....      | 68   |
| 25. An Obstruction Requirement Derived from<br>a Safety Requirement.....        | 69   |
| 26. A Repair Requirement Derived from<br>a Safety Requirement.....              | 70   |
| 27. A Privacy Requirement Derived from<br>an Achievement Requirement .....      | 71   |
| 28. A Failure Requirement Derived from<br>an Achievement Requirement .....      | 72   |
| 29. A Relaxation Lattice .....  | 76   |
| 30. Description of a Requirements Transformation<br>as an IF-THEN Rule .....    | 87   |
| 31. Examples of Transformations that are Applicable<br>and Not Applicable ..... | 89   |
| 32. A Requirements Transformation and a Sample Application.....                 | 91   |
| 33. Types of Scenario Applicability Conditions.....                             | 92   |
| 34. Examples of Changing the Initial State .....                                | 94   |
| 35. Examples of Changing Path Constraints .....                                 | 96   |
| 36. An Example of Changing the Final State .....                                | 98   |

| Figure  | Page |
|---|------|
| 37. An Example of the Display Produced by Rating Functions .....                      | 101  |
| 38. A General Scenario.....   | 110  |
| 39. A Planning Problem.....   | 111  |
| 40. A Describe Operator.....  | 116  |
| 41. Generalization of Object Types in GIRAFFE.....                                    | 119  |
| 42. A Plan Fragment Showing Possible Outcomes of Actions.....                         | 123  |
| 43. Merging Scenarios in TAMS.....  | 125  |
| 44. Indices to Plans, a la CHEF.....  | 128  |
| 45. A Summary of Some Specification Changes<br>and Requirements Transformations ..... | 191  |

## LIST OF TABLES

| Table   | Page |
|---|------|
| 1. Systems that Support Requirements Engineering .....                      | 36   |
| 2. Summary of the Systems Described in Chapter II.....                      | 49   |
| 3. Summary of Methods for Weakening and<br>Strengthening Requirements ..... | 81   |
| 4. Transformation Rules in GIRAFFE's Knowledge Base .....                   | 99   |
| 5. Rating Functions in GIRAFFE's Knowledge Base.....                        | 106  |
| 6. Requirements Transformations and Their Domain-Dependence.....            | 152  |
| 7. Rating Functions and Their Domain-Dependence .....                       | 154  |

## CHAPTER I

### INTRODUCTION

"Would you believe...?"

When faced with a difficult situation, Maxwell Smart (also known as Agent 86) often stretched the truth in hopes of convincing an adversary to set him free, or not destroy the world, or otherwise cooperate (McCrohan, 1988). Although amusing, Agent 86's fabrications were not as interesting as his subsequent unstretching of the truth—progressively weakening his story in hopes of finding something more convincing but still effective.

In a more serious (but still fictional) work (Peters, 1984), a Welsh monk describes the philosophy of the medieval Welsh legal system regarding theft. The penalty for theft was execution, which could cause a dilemma if someone were convicted of a trivial theft. The judge would have to let the offender go, and ignore the law, or carry out the sentence and thus perpetrate an offense perhaps worse than the original deed. To avoid such a dilemma, the law defined many degrees of theft, each with a correspondingly lesser penalty. If the thief stole food to stay alive, for instance, the penalty was less than it would be otherwise. By qualifying the definition of theft in this way, the law avoided the dilemma caused by extremes.

A similar problem of extremes arises in software engineering. When a statement of requirements and a specification do not agree, one has two basic choices for reconciling the two: change the specification or change the requirements.

In traditional software engineering methodologies, requirements are not changed. That approach leads to problems if a client states a requirement that is impossible to satisfy

or is unreasonably expensive. For example, a client might state that "all student schedules must be kept private." Many measures taken to ensure privacy could cause problems, such as limiting legitimate access to information, being too expensive, or being otherwise unworkable. As with the Welsh law described earlier, enforcement might be worse than violation.

Rather than changing the specification in such a case, one can change the requirements. The simplest change would be to completely discard it, but that choice still leads to problems of extremes. Instead, like Maxwell Smart and his fabrications, one can gradually weaken the requirement. One can qualify the requirement, much as the Welsh qualified their law, until it reaches a point where it avoids the problem of extremes—weak enough to be practical but not so weak that it's meaningless.

The premise of the work described in this dissertation is that changing requirements is an important alternative to changing the specification when the two conflict. In this dissertation, I describe a method of requirements transformation that supports the kind of incremental weakening necessary to solve the problems of extremes described above. The method of requirements is embodied in a program called GIRAFFE.

GIRAFFE is a program that reconciles a statement of requirements and a specification by changing the requirements. When the person using the program changes the specification, GIRAFFE responds by showing the user the strongest requirements satisfied by the altered specification. The user then has the choice of reconciling the specification either by changing the requirements, using GIRAFFE's suggestion if desired, or by further changing the specification.

By stating the strongest requirement satisfied by a specification, GIRAFFE allows its user to avoid weakening a requirement more than necessary. GIRAFFE has a set of requirements transformations which allow it to stretch— incrementally strengthen or weaken— requirements and thereby find the strongest satisfied requirement. Although the

program necessarily lacks the imagination and improvisational abilities of Agent 86, it does have the ability to find a wide range of requirements.

In order to describe GIRAFFE's techniques for requirements transformation I need to define some terms. In the remainder of this section I give informal definitions for some terms, and then in Chapter II I give more detailed definitions. Since some of these terms are used in many different ways by different writers, the definitions I give here are not necessarily standard ones.

I use the term *functional specification* to refer to the capabilities that an artifact provides. Capabilities of the artifact allow agents to perform various actions. For example, an on-line registration system might provide the capability for a student to hear her schedule over the phone. Names of capabilities are enclosed in curly braces, such as {phone list schedule}.

The term *requirements* in this dissertation refers to a high-level description of an artifact that, unlike the functional specification, is not stated in terms of capabilities and actions. Requirements are stated as state transitions that should or should not occur in a given artifact or in the environment of the artifact. For instance, a requirement might say that there should be a transition to go from a state where a student does not know her schedule to a state where she does know her schedule. Requirements can require (describe as desirable) or prohibit a transition, and they can describe the way the transition should occur (e.g., within two seconds). Names of requirements are enclosed in angle brackets, such as <find out schedule>.

I use the term *analyst* as a generic term that can refer either to a human analyst or to a program like GIRAFFE that assists with requirements engineering but has no authority to change requirements. I use the term *client* to refer to the person who can authoritatively state and change requirements for the artifact. I also refer to GIRAFFE's user as a client, to avoid confusion with users of the artifact.



In the next section of this chapter I give an example of requirements transformation and a short description of the GIRAFFE program. In the third section I give three reasons why requirements transformation is important. Then I state the thesis, assumptions, and contributions of this work. I conclude the chapter by outlining the remaining chapters of the dissertation.

### Requirements Transformation

In this section I describe a model of requirements engineering in terms of requirements transformations. In this model, the client begins by describing the functionality of a proposed artifact. The analyst responds by indicating how well the artifact satisfies the requirements. The client then suggests additional functional changes or accepts transformed requirements.

Requirements transformations play two roles in this process. Changes in requirement satisfaction can be expressed as requirements transformations because the analyst describes satisfaction by stating the strongest requirement satisfied by the changed functional specification. The difference between this new strongest satisfied requirement (or just "satisfied requirement") and the previous satisfied requirement can be expressed as a requirements transformation. Likewise, the difference between a satisfied requirement and the original requirement (or "target requirement") can also be stated as a requirements transformation.

First I give an informal example of requirements transformation. Then I describe GIRAFFE's role in requirements engineering.

### An Example

Figure 1 shows a hypothetical conversation from the domain of on-line registration at a university. At the start of the session shown in Figure 1, the analyst has in mind a set

of requirements for an on-line registration system. The requirements that are relevant for this example are "a student should be able to find out her own schedule" and "no one else should be able to find out a student's schedule" (see Figure 2). Those requirements could have been stated earlier by the client, or they could be from a standard set which the analyst knows from past experience.

The client begins by describing a change to the functional specification: adding a {list schedule} capability. The analyst responds by indicating how well the changed specification satisfies the requirements. The specification provides a way for students to find out their schedule but only on the condition that they have access to the system (i.e., they have phone access and know the registration system's phone number). The analyst also points out that <schedule privacy> will be affected because intruders that have access to the system will be able to find out a student's schedule. Satisfaction of <schedule privacy> after the change will be weaker, or, put another way, the satisfied requirement will be weaker.

The conversation continues, with the client suggesting other functional changes, such as requiring a password to list a schedule, and adding a capability for assigning and distributing passwords. After each change, the analyst describes changes in satisfaction due to the change. The client responds by changing the specification, as with the addition of a password to {list schedule}, or by changing the requirements.

Changes in the requirements are indicated in Figure 1 by the client acknowledging that a requirement will not be fully satisfied. For instance, when informed that adding a password will weaken satisfaction of <find out schedule>, the client acknowledges the weakening, thus effectively changing the requirements. Such a change is referred to in this section as "transforming the target requirements" but is referred to in later sections simply as "transforming the requirements."

CLIENT: Students should be able to list the classes they're registered for.

ANALYST: We'll add functionality to the system for students to list their classes by phone. With that change, students will be able to find out their schedules as long as they have access to the system. A student can find out another student's schedule, though.

CLIENT: Have the system ask for a password before listing the schedule.

ANALYST: That will eliminate some ways of finding out another student's schedule. However, because of the password check, listing schedules will require one more action and will take longer.

CLIENT: That's OK.

ANALYST: Students have to know their own passwords to find out their own information. There has to be a way to assign passwords and a way for students to find out their passwords.

CLIENT: Students will be assigned random passwords that are given to them when they show ID.

ANALYST: Random passwords are more difficult to guess but are also more difficult to remember. If a student forgets her password she won't be able to find out her schedule information, and students will have to go to the registrar's office to get their passwords. Even with random passwords, an intruder can find out someone else's schedule if a student tells her password.

CLIENT: Students are responsible for not telling their passwords.

ANALYST: OK. Although it is more difficult, an intruder can still find out a random password.

CLIENT: Students can change their passwords if someone finds out.

FIGURE 1. A hypothetical conversation between an analyst and a client.

To see the role of requirements transformations in this conversation, consider the first change made by the client in Figure 1: adding {list schedule}. The change in satisfaction of <find out schedule> after that change is illustrated in the left half of Figure 3.

The analyst compares the version of <find out schedule> satisfied before the change to the version satisfied after the change. In this case, the requirement before the change is the weakest form of <find out schedule> because the specification doesn't include any capability that students can use to find out schedules.

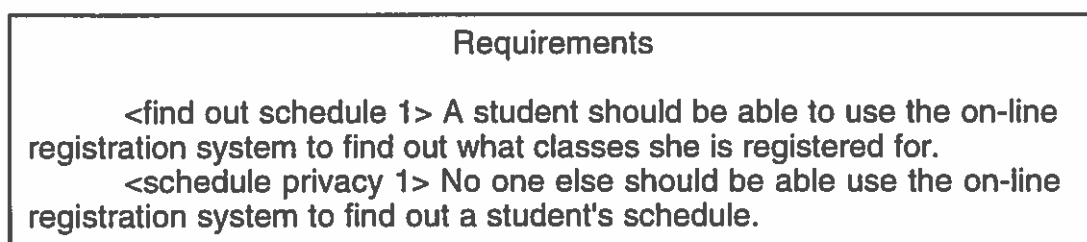


FIGURE 2. Informal statements of two requirements for an on-line registration system.

The requirement satisfied after the change is a stronger requirement because the change makes it possible for a student to find out her schedule where it wasn't possible before. T1 in Figure 3 is a requirements transformation that describes the difference in satisfied requirements before and after the functional change.

The analyst also describes how the requirement satisfied by the new specification falls short of the original, or "target" requirement. In this case, a student can find out her schedule only if she has access to the registration system, so the difference is the new condition that the student must have access to the system.

To reconcile the difference between the satisfied requirement and the target requirement, the client can explicitly weaken the target requirement as shown in the right half of Figure 3. T2 is a requirements transformation that makes <find out schedule> weaker and in this case makes the target requirement the same as the satisfied requirement.

The process continues, with the analyst using transformations to describe changes in satisfaction and to suggest changes to the target requirements. When the satisfied

requirements and the target requirements agree, the requirements engineering process is finished.

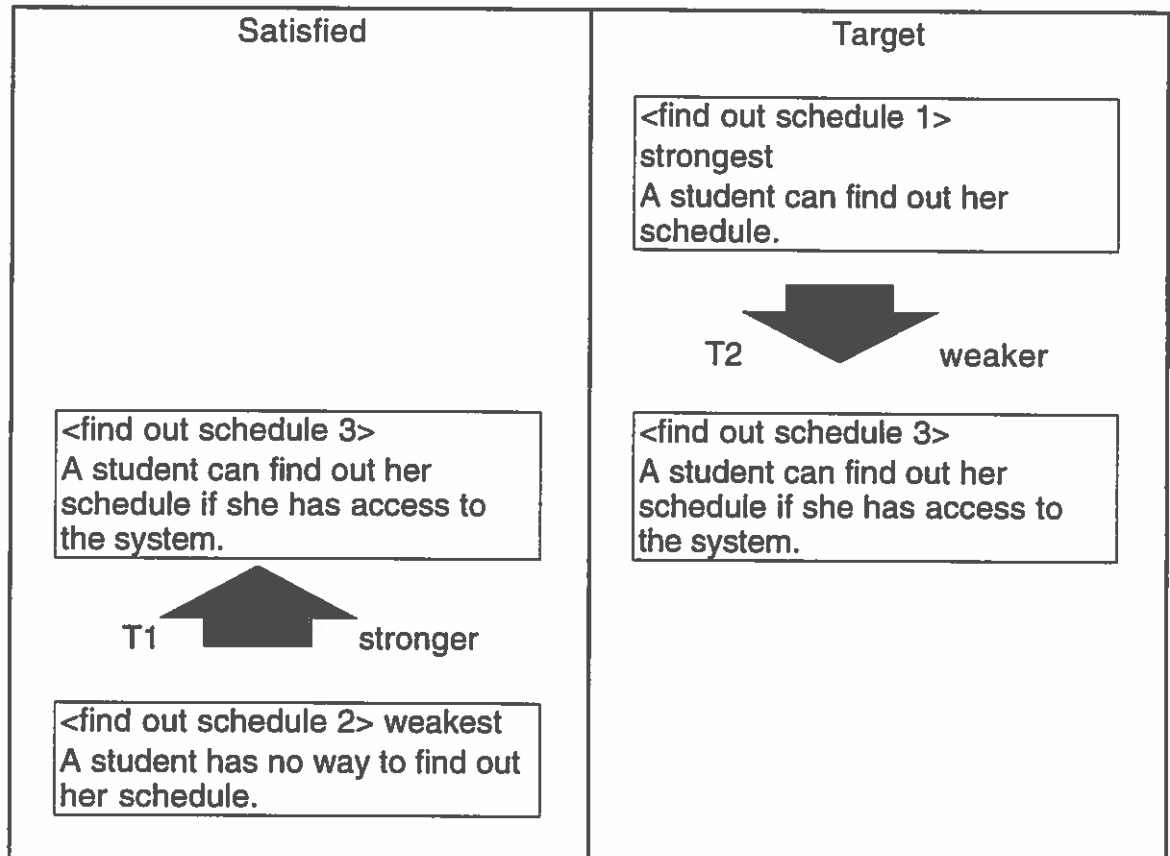


FIGURE 3. Transformation of satisfied and target requirements. T1 represents a transformation of the requirement satisfied by the specification. T2 represents a transformation of the target requirement.

### GIRAFFE

I have created a program, called GIRAFFE, that automates parts of this process of requirements engineering. GIRAFFE takes the role of the analyst in Figure 1. The program allows the client to describe functional changes to the artifact's specification and then describes to the client changes in the satisfied requirements. In describing changed requirements to the client, GIRAFFE states the strongest requirements satisfied by the

current specification. The client can then indicate whether the satisfied requirements are acceptable or not. If they are acceptable, GIRAFFE changes the target requirements; if not, the client can further modify the functional specification.

At the beginning of a requirements engineering session, GIRAFFE might start out with an ideal version of the <find out schedule> requirement (like <find out schedule 1> in Figures 2 and 3) as a target requirement. In <find out schedule 1> a student can find out her schedule without qualification. GIRAFFE's domain model includes a standard set of requirements, like <find out schedule 1>, which the program can transform in response to the client's input.

Figure 4 shows various requirements that GIRAFFE could produce as a result of requirement transformations. Requirements toward the top of the figure are stronger than those toward the bottom of the figure. The ideal version of <find out schedule> is shown at the top of Figure 4.

The client can begin with a null specification or can begin with a specification produced by some other process. If the initial specification has no capability for a student to find out her schedule, the requirement initially satisfied by the specification is the weakest version of <find out schedule>, shown at the bottom of Figure 4 and labeled "no way to find out schedule."

The client's first action is to change the specification. If the client adds a capability for finding out schedules by phone to the specification, as in Figure 1., then GIRAFFE weakens the initial, unrealistic requirement to the "phone" requirement shown on the left side of Figure 4. Unlike the ideal requirement, the "phone" requirement has a qualification: a student must have access to the system (know the phone number and have phone access) to find out her schedule. Adding qualifications such as this one to a requirement make it weaker.

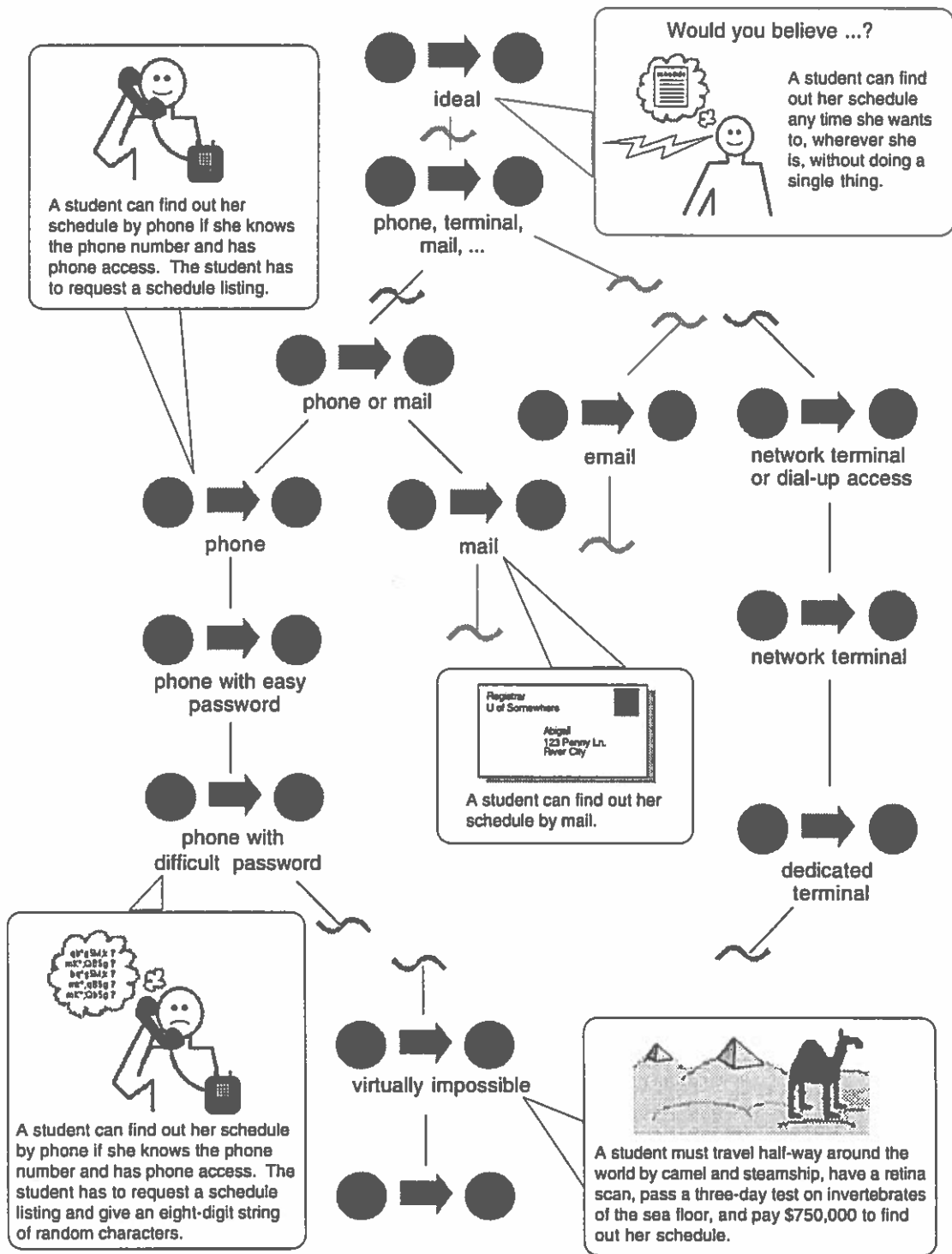


FIGURE 4. Parts of a requirements lattice. Two circles with an arrow between represent different versions of the <find out schedule> requirement. Squiggles represent elision of parts of the lattice.

The "phone" requirements in Figure 4 differ from the "mail" requirement (shown in the center of the figure) because when finding out her schedule by phone a student must explicitly request a schedule listing. When a student is required to do an action, a requirement is weaker than it would otherwise be because there is a qualification: the requirement is only satisfied under certain conditions, in this case when the student performs a certain action.

Another example of a change in requirements occurs in Figure 1 when the client specifies that students must give a password to get a schedule listing. When the client makes that specification change, GIRAFFE responds by transforming the requirement from the phone requirement in Figure 4 to "phone with difficult password". This change corresponds to the analyst telling the client that a student must know her password and that it will take students longer to find out their schedules because of the check. The change in time required is an example of a change in an attribute of a requirement, where the attribute concerned is duration.

GIRAFFE does not construct an explicit lattice of requirements as shown in Figure 4. Instead it uses transformations to make changes that correspond to moving from one point in the lattice to another. In Chapter IV I describe GIRAFFE's knowledge base of transformations. GIRAFFE's transformations describe changes in requirements in terms of attributes (like duration), generality of objects, and qualifications (like a student needing access to the system). The transformations are based on a relation between requirements called IS-STRONGER-THAN that is defined in Chapter III.

In this section I have described a model of requirements engineering and have discussed a program's role in that process. I began with a hypothetical, informal description of one example (Figure 1) and then showed the role that requirements transformations play in that example (Figure 3). The requirements in Figure 2 are informal and the transformations in Figure 3 are shown only as the difference between two informal



requirements. For GIRAFFE to act as the analyst it needs a formal representation of requirements and needs to formally represent and reason about transformations like the ones in Figure 3. In later chapters of this dissertation I examine a formal representation for requirements and requirements transformations. Next I look at the motivation for creating the GIRAFFE program.

### Why Transform Requirements?

In this section I give several reasons for creating a system like GIRAFFE. I describe how GIRAFFE will support intertwining in the requirements engineering process, provide useful evaluation information, and support reuse of requirements and components.

#### Intertwining

Swartout and Balzer (1982) use the term *intertwining* to describe a relationship between specification and implementation. They argue that it is impossible to completely separate specification and implementation in software development because physical limitations and lack of foresight (i.e., an insufficient model of the artifact in its environment) will cause specification modifications. They conclude that software development tools should address the interleaving between specification and implementation rather than keeping specification and implementation completely separate.

GIRAFFE addresses intertwining between requirements and specifications. Swartout and Balzer indirectly describe this kind of intertwining when they argue that "every specification is an implementation of some other higher level specification" (Swartout & Balzer, 1982, p. 438). The same factors that make modifications to the specification necessary make modifications to the requirements, a "higher-level specification," necessary. GIRAFFE helps clients transform requirements when limitations or lack of foresight make those transformations necessary.

### Evaluation

GIRAFFE supports intertwining and helps the client state requirements that reflect the limitations of an artifact's specification. When the client states requirements that cannot be satisfied, GIRAFFE describes a weaker requirement that can be satisfied, rather than simply saying that the requirement is not satisfied. Similarly, if a specification satisfies a stronger requirement, GIRAFFE tells the client so that the requirement can be transformed. By giving this type of evaluation, GIRAFFE helps the client state requirements and specifications that recognize limitations and describe more robust artifacts where possible.

### Reuse

Since GIRAFFE helps the client transform requirements, it allows requirements to be stated generally and then adapted to a particular situation. Thus it promotes reuse of requirements.

GIRAFFE also promotes reuse of components by transforming requirements to include characteristics of available components. While it is possible to begin with requirements and then find components that satisfy those requirements, Lubars, Potts, and Richter (1993) describe in a survey of requirements engineering practice a situation where the order was changed: "The customer wanted to see a requirements model that includes properties and behavior of the off-the-shelf components, because integrating the components was the riskiest aspect of the project" (Lubars et al., 1993, p. 10).

If components exist to satisfy a set of requirements, then the traditional process of requirements engineering can lead to reuse of components. However, if no components satisfy the requirements than the traditional process will not support reuse since it doesn't allow specification to influence requirements. By helping the client to weaken requirements

acceptably, GIRAFFE promotes reuse of components by allowing specification to influence requirements.

### Thesis Statement

My thesis is that knowledge-based analysis of environmental constraints and requirements relations supports requirements transformation in software engineering by allowing a program to find the strongest requirement satisfied by a specification. Finding the strongest satisfied requirement avoids unnecessary weakening and supports intertwining, evaluation and reuse in requirements engineering.

"Environmental constraints" refers to actions provided by the environment, conditions that might arise in the environment—initial conditions—and the kinds of actions that agents in the environment might or might not perform—path constraints.

"Requirements relations" refers to comparing requirements in terms of strength and weakness, and to other relations between requirements such as plan support/obstruction and decomposition.

### Assumptions

My work rests on two basic assumptions. One assumption addresses the nature and extent of intertwining and the other addresses the representation for requirements and functional specifications used by GIRAFFE.

In my research I will not study in detail the current practice of requirements engineering. Other researchers have discussed intertwining and related issues (Reubenstein, 1990; Swartout & Balzer, 1982; Lubars et al., 1993) but the evidence is basically anecdotal. I am assuming that the problem exists to a degree that makes this work significant. I also assume that the client, in a significant number of instances, is familiar

with the options for the functional specification so that requirements transformation is a useful approach.

In defining GIRAFFE's transformations I have assumed that the client wants achievement requirements to occur in as many situations as possible and safety violations to occur in as few situations as possible. Other constraints are possible. For instance, a client might not care how many violations of some safety requirement occur after the first. The IS-STRONGER-THAN relation does not apply to constraints for which this assumption does not hold.

This work relies on representation of requirements as transitions from an initial state to a final state with various path constraints added to describe additional functional requirements and non-functional requirements. There is some indication (Anderson & Durney, 1993) that this is a useful representation for requirements but my work does not directly address this issue.

### Contributions

In this dissertation, I define a method of requirements transformation that addresses intertwining, provides effective evaluation and supports reuse of requirements and components. In defining and evaluating that method I make two contributions. First, I define a set of requirements relations, including one called IS-STRONGER-THAN that allows a program to compare the strength of alternative requirements. Second, I describe a method for finding general scenarios.

In defining the requirements relations described in this dissertation, I have applied work in dimensions to requirements engineering and extended it by defining a stronger-than relation for requirements. The requirements transformation method supports analysis in terms of requirement subsets, attributes, object type generality, and qualifications, and

so provides a finer granularity than is possible using only scalar dimensions (Rissland, 1986) or subsets of requirements (Herlihy & Wing, 1991).

The requirements transformation method also includes relations used to derive related requirements, which is a contribution to requirements acquisition and definition because the relations allow a program to suggest new requirements. These relations are an adaptation of work in goal relations in planning (Carbonell, 1981; Wilensky, 1983) to requirements engineering.

In defining the method for finding general scenarios described in this dissertation, I have extended Anderson's work on OPIE (Anderson & Farley, 1988, 1990). The method I define uses a kind of general plan that allows a program to describe partial satisfaction of requirements and satisfaction of stronger requirements. GIRAFFE's planner (a modified version of OPIE) can find more general plans and more specific plans that relate to a given planning problem. The planner can work with partially specified initial states and a domain model to find plans when no plan is possible for the original initial state. The planner also finds multiple paths.

### Overview

In the next chapter I describe GIRAFFE's representation for requirements, specifications and scenarios and give an overview of the process by which GIRAFFE transforms requirements. I compare GIRAFFE to other requirements engineering systems.

In Chapter III I define the IS-STRONGER-THAN relation which GIRAFFE uses to compare requirements. I also describe relations that GIRAFFE uses to derive new requirements. I compare GIRAFFE's analysis of requirements to other work that deals with stronger and weaker goals and with other other work that deals with deriving new requirements.

I describe the transformations and rating functions that make up GIRAFFE's knowledge base in Chapter IV. I show how the knowledge base incorporates the IS-STRONGER-THAN relation defined in Chapter III and how it is used in the process described in Chapter II.

In Chapter V I describe GIRAFFE's method for finding scenarios. GIRAFFE uses a modified form of OPIE (Anderson, 1993) that can find general scenarios. I describe the general scenarios and why they are important for requirements generation. I compare GIRAFFE's method of finding scenarios to the methods used by other programs.

In Chapter VI I evaluate GIRAFFE's performance in the domain of on-line registration. I summarize GIRAFFE's derivation of requirements for three different artifacts and summarize comments on GIRAFFE given by an analyst and a client. I also discuss GIRAFFE's ability to analyze requirements in other domains. I indicate which transformations and rating functions in GIRAFFE's knowledge base are domain-dependent and the degree to which they are domain-dependent.

In the concluding chapter I summarize GIRAFFE's contributions and discuss future work. Future work includes additional methods of finding scenarios, automated support for building the domain model, and textual descriptions of scenarios.

## CHAPTER II

### GIRAFFE'S MODEL AND PROCESS

#### Introduction

GIRAFFE improves requirements engineering by supporting intertwining and reuse and by evaluating specifications in terms of partial satisfaction of requirements. In this chapter I give an overview of how GIRAFFE provides these improvements. First I describe the representation that GIRAFFE uses, and then the process by which it transforms requirements. Finally I compare GIRAFFE to other requirements engineering systems to show how it realizes advantages that other systems do not.

#### Representation

GIRAFFE's representations of requirements and specifications are important because the program's output is stated as changes in requirements and its input is stated as changes in the specification. GIRAFFE's representation of scenarios is important because scenarios are the element of GIRAFFE's model that allows it to relate specifications to requirements.

Actions and conditions are common elements of GIRAFFE's representation of requirements, specifications, and scenarios. Specifications are composed of action types called capabilities, and scenarios are sequences of action instances. Conditions are used in defining requirements and scenarios.

In this section I define the elements of GIRAFFE's model: requirements, specifications, scenarios, capabilities and actions, and conditions. For each element of

GIRAFFE's model I give an informal and formal definition. The definitions in this section are also in the glossary in Appendix A.

I begin by defining requirements. As part of the definition of requirements, I define transitions and qualifications, two important components of requirements. Next I define specifications in terms of capabilities and capability sets, and finally I define scenarios.

### Representation of Requirements

GIRAFFE represents requirements as transitions from an initial state to a final state with qualifications. GIRAFFE's representation for requirements is similar to the one used by OPIE (Anderson & Durney, 1993) except that GIRAFFE's representation also includes qualifications such as path constraints. A path is a sequence of actions and intermediate states that show how a transition can occur, and path constraints state what actions and intermediate conditions can occur in that sequence. Figure 5 shows the initial state, final state and path constraint for a requirement that a student should be able to find out her class schedule without going to the registrar's office.

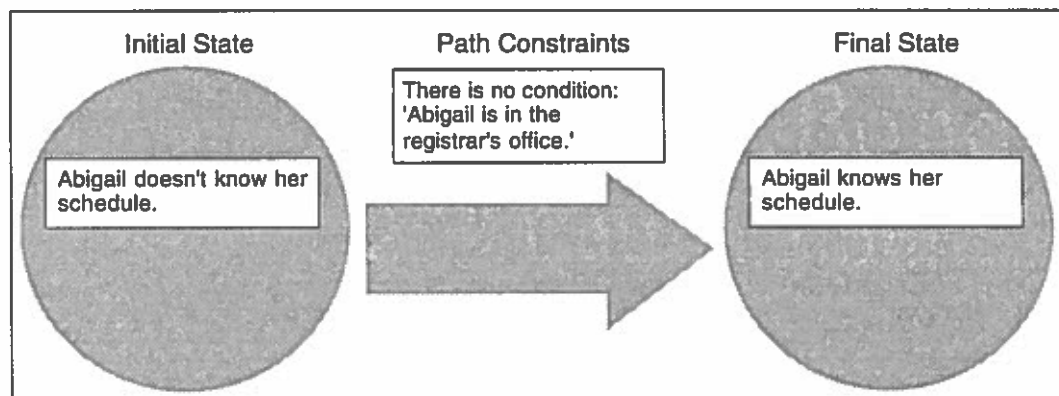


FIGURE 5. A transition that represents an achievement requirement called <find out schedule>.



Formally, a *requirement* is a tuple  $\langle name, type, O, trans, Q, A, Related \rangle$ , where *name* and *type* are symbols and *O* is a set of object definitions. An *object definition* is a pair  $\langle obj\_type, obj\_name \rangle$ , where *obj\_type* and *obj\_name* are symbols. *Obj\_type* is a symbol representing a type defined in an object-type hierarchy.

In the requirement tuple, *trans* is a state transition, *Q* is a set of qualifications, and *A* is a set of attributes. *Related* is a set of symbols, each representing the name of a related requirement. When GIRAFFE derives new requirements it keeps track of the relations between a parent requirement and the children requirements derived from it. In Chapters III and IV I describe GIRAFFE's methods for deriving requirements and how relations between derived requirements affect other requirement relations.

A *transition* is a tuple  $\langle IC, PC, FC, Scenarios \rangle$ , where *IC* and *FC* are sets of conditions, *PC* is a set of path constraints, and *Scenarios* is a set of scenarios. Transitions are a useful representation for requirements because GIRAFFE can use them to determine requirement satisfaction. If there is a scenario, or sequence of actions, showing how the transition can occur, then an achievement requirement is satisfied. If there is no scenario showing how a final state can occur, given the initial state, then a safety requirement is satisfied.

The initial state and final state list the conditions that define a given transition. A *condition* is a pair  $\langle relation, objects \rangle$  where *relation* is a symbol and *objects* is a list of symbols representing the objects that play roles in the relationship. Conditions are relations that hold between objects at a certain time. For example, the initial condition in Figure 5 represents a relation between a student, Abigail, and some information. State descriptions list only the relevant conditions. Other conditions will also be true but they are not relevant for determining whether or not the transition has occurred.

Path constraints are statements about the actions and intermediate conditions that occur during execution of the transition. For example, the transition in Figure 5 has a path

constraint that Abigail is not at the registrar's office. Path constraints can also refer to the number of paths from the initial state to the final state, and they can refer to the duration of one or more paths.

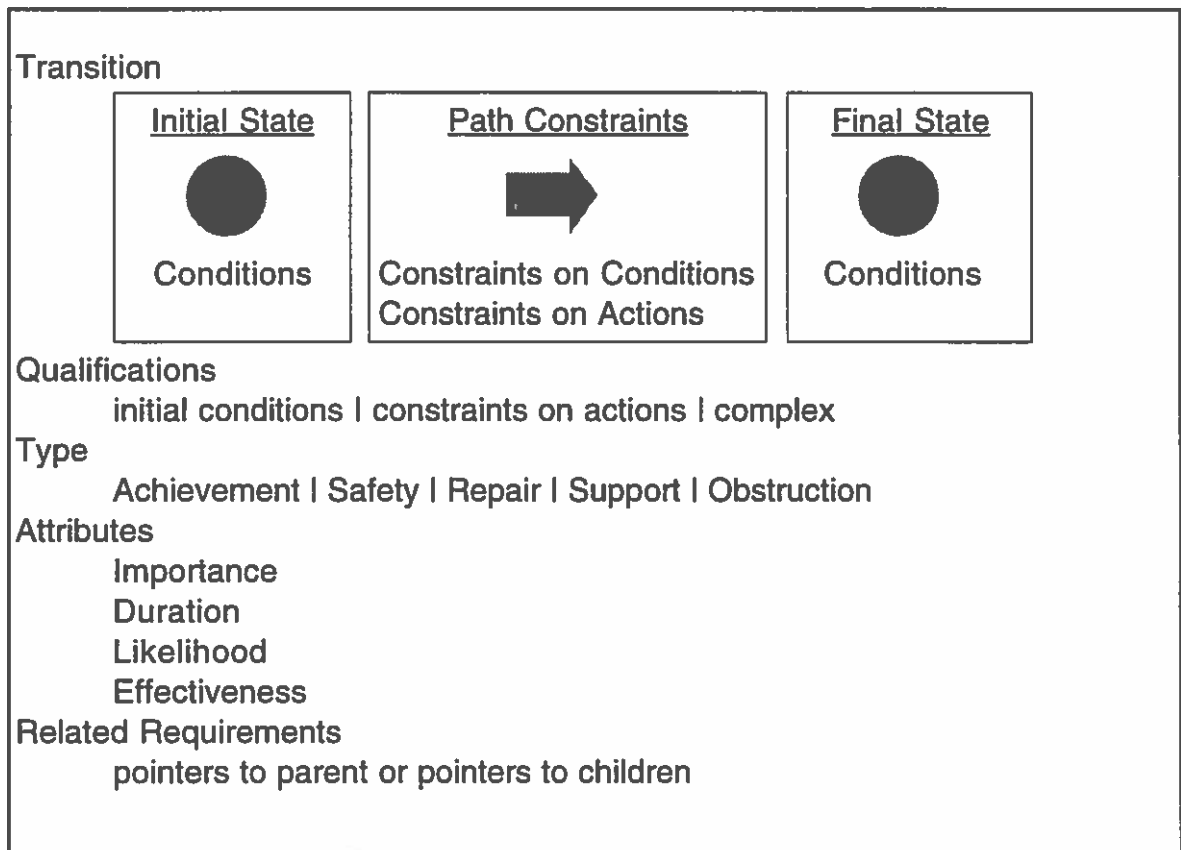


FIGURE 6. Summary of GIRAFFE's representation of requirements.

Requirements can have attributes including importance, duration, likelihood, and effectiveness. An *attribute* is a pair  $\langle attr\text{-}name, attr\text{-}value \rangle$ , where *attr-name* is a symbol, and *attr-value* is a number or a symbol. GIRAFFE uses relative or qualitative values for attribute values rather than absolute numerical values. All of the attributes except importance are derived from corresponding attributes of scenarios associated with the requirement. Such attributes can be broken down into minimum, average and maximum values. Some attributes for scenarios are derived from attributes of the actions in the

scenario. For example, the duration of a scenario is the sum of the duration attributes of actions.

Figure 6 summarizes GIRAFFE's representation of requirements. In the next section I explain the types of requirements shown in Figure 6. Then I define qualifications and describe how the conditions and constraints in Figure 6 can be considered definitions or qualifications.

### Types of Requirements

There are two general types of requirements in GIRAFFE: achievement requirements and safety requirements. Achievement requirements are transitions that the client wants the artifact to support or allow, and safety requirements are transitions that the client wants the artifact to disable or discourage. The transition in Figure 5 represents an achievement requirement while the transition in Figure 7 represents a safety requirement. Other types of requirements are repair requirements, support requirements and obstruction requirements.

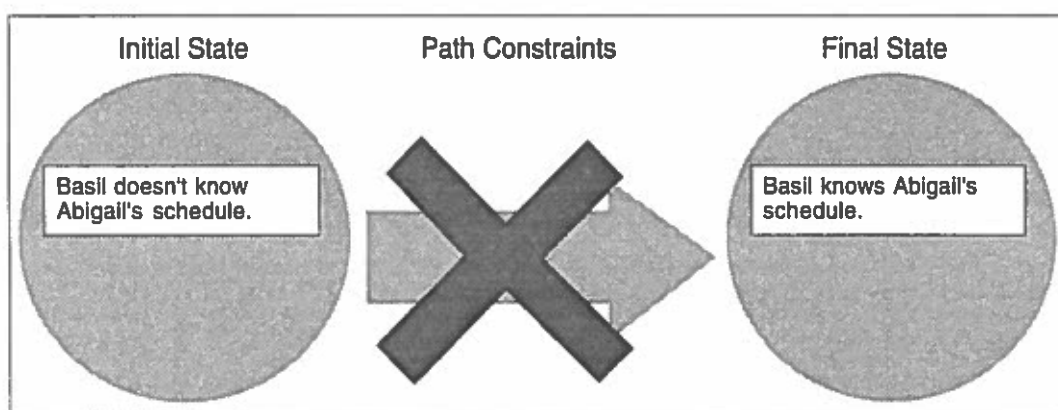


FIGURE 7. A transition that represents a safety requirement called <schedule privacy>.

Repair requirements are similar to achievement requirements in that they show achievement of a desired state. They differ in that their initial conditions are not desirable, whereas the initial conditions of achievement requirements can themselves represent a desired state. An example of a repair requirement is that if an intruder knows a student's password there is a way to reach a state where the intruder doesn't know the student's password. The initial state includes an undesired condition, namely that a password is not secure.

A support requirement is a type of achievement requirement that derives its value from the value of some other achievement requirement. For example, if it is only possible for a student to find out her schedule if she knows her password then a support requirement could be that it is possible for a student to find out her password. If <find out schedule> had no value then the support requirement <find out password> would also have no value.

Obstruction requirements also derive their value from another requirement, but from safety requirements rather than achievement requirements. For instance, suppose that <schedule privacy> can be violated in a certain situation if an intruder knows a student's schedule. Then <password privacy> is an obstruction requirement because it shows a way of preventing a safety violation. Just as support requirements are a subset of achievement requirements, obstruction requirements are a subset of safety requirements.

The type of a requirement is important in determining satisfaction of the requirement. An achievement requirement is satisfied if there is at least one scenario showing a transition from the initial state to the final state. A safety requirement is satisfied if there are no scenarios from the initial state to the final state.

The type of requirement is also important in deriving new requirements. A support requirement can be derived from an achievement requirement but not from a repair

requirement. Likewise, repair requirements are derived from safety requirements and not achievement requirements. Chapters III and IV discuss derivation of related requirements.

### Qualifications

Qualifications represent assumptions that must hold for a requirement to be satisfied. For achievement requirements, qualifications can be assumptions that some conditions must be true in the initial state or that some action must or must not occur in order for a transition to occur. For safety requirements, qualifications can be assumptions that some initial conditions must not be true, or that some action must or must not occur in order to prevent a violation from occurring.

Figure 8 shows a requirement for an artifact that includes an on-line advisor. The basic requirement is that students can get advice from the artifact, and the figure shows examples of how the initial state and the path can be qualified. Line 1 is a qualification of the initial state; it describes a condition that must hold for the transition to execute. Line 2 is a path constraint; if a specified action occurs (the student forgets her password) then the transition cannot execute. While it is possible to qualify the final state of a transition in addition to the initial state and the path, GIRAFFE does not use qualifications on the final state.

|  |
|--|
| <p>A student can get advice from the artifact</p> <p>1 if the network is up<br/>[has_init (has_access_to (online_reg reg_db))]</p> <p>2 unless she forgets her password<br/>[does_not_have_action forget_password]</p> |
|--|

FIGURE 8. Two qualifications for a requirement. A formal statement of each qualification, in brackets, follows each informal statement.

Some qualifications can be stated in terms of responsibilities (Feather 1987; Fickas and Helm 1991): "no student can find out another student's password unless a student doesn't fulfill her responsibility and tells her password."

The formal definition of qualifications in GIRAFFE is as follows:

Definition 1

A *qualification*,  $q$ , is defined as follows:

$$q ::= q_{prim} \mid q_{complex}$$

$$q_{prim} ::= q_{init} \mid q_{action}$$

$$q_{init} ::= \langle q\_type, condition \rangle, \text{ where } q\_type \text{ is the symbol } has\_init.$$

$$q_{action} ::= \langle q\_type, action\_type\_name, agent\_type\_name \rangle \mid$$

$\langle q\_type, action\_type\_name \rangle$  where  $q\_type$  is one of the symbols  $has\_action$ ,  $does\_not\_have\_action$ , or  $env\_action$ .

$$q_{complex} ::= q_{complex} \vee q_{term} \mid q_{term}$$

$$q_{term} ::= q_{term} \wedge q_{prim} \mid q_{prim}$$

$Action\_type\_name$ , and  $agent\_type\_name$  are symbols. □

Qualifications of the form  $q_{action}$  represent path constraints added to weaken a requirement. Qualifications of the form  $q_{init}$  represent assumptions about the initial state of the transition. Logical expressions made up of path constraints and initial conditions make up qualifications of the form  $q_{complex}$ .

A condition in the initial state of a transition can be part of the definition or it can be a qualification. Definitions describe an interesting transition and qualifications describe the situations in which a transition can occur. For example, if a requirement is that a student should be able to drop a class, one condition in the initial state is that the student has a class in her schedule. If that condition is removed from the initial state the transition changes and is no longer of interest.

Conditions that define a transition instead of qualifying it are especially important for repair requirements. For example, a repair requirement might state that if a student's address is incorrect in the database there must be a way to correct it. An initial condition in that requirement is that the student's information is incorrect. This condition defines the transition rather than making it weaker as a qualification would.

Another condition that might be in the initial state of the <drop class> requirement is that the student has phone access. Whereas it makes no sense to drop a class if the student doesn't have it in her schedule, it does make sense to drop a class without having phone access.

The distinction between definitions and qualifications is important because GIRAFFE will transform qualifications but not definitions. A condition or constraint that is a definition might be a qualification in another requirement. The client can state that conditions or constraints are definitional. In some cases GIRAFFE infers that a condition is definitional, such as in a repair requirement.

### Representation of a Specification

GIRAFFE represents a specification as sets of capabilities. Each capability represents an action that some agent can perform. In this section I describe capabilities and the various capability sets that GIRAFFE uses.

#### Capabilities

GIRAFFE represents a capability as a means of changing the conditions that hold in a situation. Figure 9 shows an example of a capability called {phone drop class}. If that capability is part of an artifact, and an agent uses the capability to perform an action, then the changes in conditions listed in the figure will occur.

**Definition 2**

A *capability* is a tuple  $\langle name, agent, otypes, consumed, used, produced, Super, Sub \rangle$  where *name* is a symbol, *agent* is a symbol representing an object type, *otypes* is a list of symbols representing object types, and *consumed*, *used*, and *produced* are sets of conditions. *Super* and *Subs* are sets of symbols, where each symbol represents an action type, and are part of a hierarchy of action types. □

Some conditions, such as the condition that a certain class is in a certain schedule, will be consumed when an agent uses a capability to perform an action. Consumed conditions are required to hold before the action occurs and do not hold after the action occurs. Used conditions are required to hold before the action occurs. They still hold after the action occurs. The condition that a certain program can do phone I/O is used during execution of {phone drop class}. Produced conditions hold after the action occurs. The condition that a seat is available for a certain class holds after {phone drop class} occurs.

Each condition is a relationship among objects. The objects in Figure 9 are represented as instances of object types. For instance, *person1* is a certain instance of object type *person*. Object types are important for GIRAFFE because it uses a type hierarchy to reason about changes in generality of requirements.

Each capability represents an action that a particular kind of agent can perform. For the capability in Figure 9, the agent is *program1*. The type of agent is important because some capabilities represent actions that are controlled by an agent that is part of the artifact, and some capabilities represent actions that are controlled by an agent in the environment. The *art\_agent* attribute in Figure 9 indicates that this capability represents actions that are controlled by the artifact.



```

name: phone drop class
description: program drops class for a student
objects: program1 person1 person2 term1 class1 seat1
sched_info1 databasel

consumed conditions:
person2 requested a drop for class1 in term1
class1 is in sched_info1

used conditions:
program1 can do phone I/O
person2 has a phone connection to program1
sched_info1 is in databasel
sched_info1 is the schedule information for person2
for term1
program1 has access to databasel

produced conditions:
seat1 is available for class1 for term1
class1 is not in sched_info1

attributes: art_agent

```

---

```

(def_etyp "phone drop class"
:d "program drops class for a student"
:o '(program1 person1 person2 term1 class1
    seat1 sched_info1 databasel)
:- '((ph_requested_drop (person2 class1 term1))
    (class_in_sched (class1 sched_info1)) )
:= '((can_do_phone_IO (program1))
    (has_phone_connection (person1 person2 program1))
    (has_schedule_info (person2 term1 sched_info1))
    (db_has_s_in (databasel sched_info1))
    (has_access_to (program1 databasel)) )
:+ '((seat_available (seat1 class1 term1))
    (not_class_in_sched (class1 sched_info1)) )
:at '((category :val art_agent))
)

```

FIGURE 9. An operator that represents a capability of an artifact. An informal description is above the line, and the formal definition used by GIRAFFE is below the line.

The formal representation of capabilities, exemplified in the bottom part of Figure 9, enables a program such as a planner to reason about what scenarios can occur if a particular artifact is in a particular environment.

### Capability Sets

GIRAFFE uses three kinds of capability sets: one representing the environment of the artifact, another representing the current functionality of the artifact, and a third representing potential functionality of the artifact. When GIRAFFE calls the planner to find scenarios, it gives the environment capability set and the artifact capability set to the planner as the planner's operator set (Anderson & Fickas, 1989).

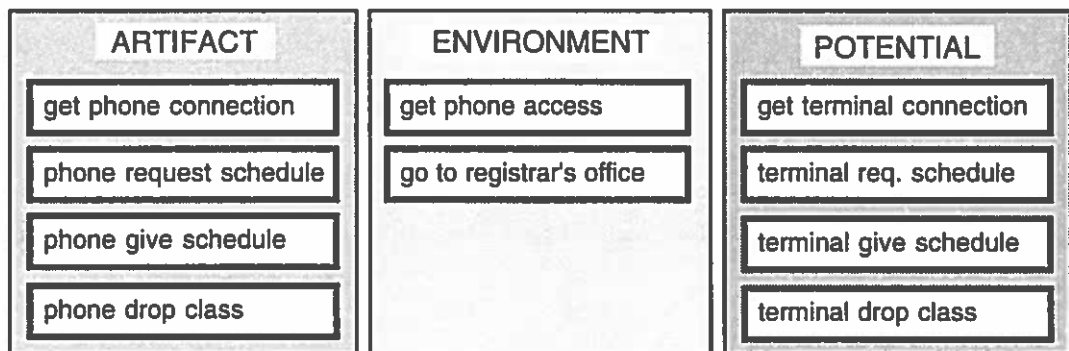


FIGURE 10. Examples of the three capability sets used by GIRAFFE.

Figure 10 shows the three kinds of capability sets with examples of capabilities. In the figure, the artifact has the capability to allow people to get phone connections and request schedule listings. The artifact also has the capability to give schedules over the phone and to drop classes when requested by phone. Note that some capabilities in the artifact set represent actions performed by agents in the environment (e.g., students) if those actions would not be possible without the artifact. Actions that are performed by agents in the environment and that would be possible without the artifact are represented by

capabilities in the environment capability set. Appendix B gives an example of an environment capability set.

Actions that could be made possible by the artifact, but aren't in the current specification, are represented by capabilities in the potential capability set. For example, in Figure 10 the program could process requests to drop classes entered at a terminal, but that capability is not part of the currently specified artifact. Appendix C gives examples of capabilities that appear in the artifact capability set and the potential capability set.

During requirements analysis, the client moves capabilities from the potential capability set to the artifact capability set, or vice versa, and GIRAFFE analyzes the change in requirement satisfaction caused by the change. A later section of this chapter describes GIRAFFE's process for analyzing such changes.

### Representation of Scenarios

Scenarios are important because they allow GIRAFFE to relate specifications to requirements. They describe how transitions can occur given the specification of an artifact and a description of its environment.

A scenario is a partially-ordered set of actions. An *action* is a tuple  $\langle \textit{capability}, \textit{agent}, \textit{objects}, \textit{consumed}, \textit{used}, \textit{produced} \rangle$  where *capability* is a symbol representing the capability required for the action, *agent* is an object definition, *objects* is a list of object definitions, and *consumed*, *used*, and *produced* are sets of conditions.

A scenario shows how a given final state can be achieved from a given initial state. Scenarios include causal links that show which conditions are required by an action instance and which conditions an action instance produces. Each action includes references to objects that affect or are affected by the action, including the agent of the action.

Figure 11 shows an example of a scenario. The scenario has three actions, each of which requires conditions and produces conditions, although not all of the conditions are

shown in the figure. The actions in the scenario are instances of capabilities from the artifact capability set or the environment capability set. This scenario shows how the transition shown in Figure 5 could occur.

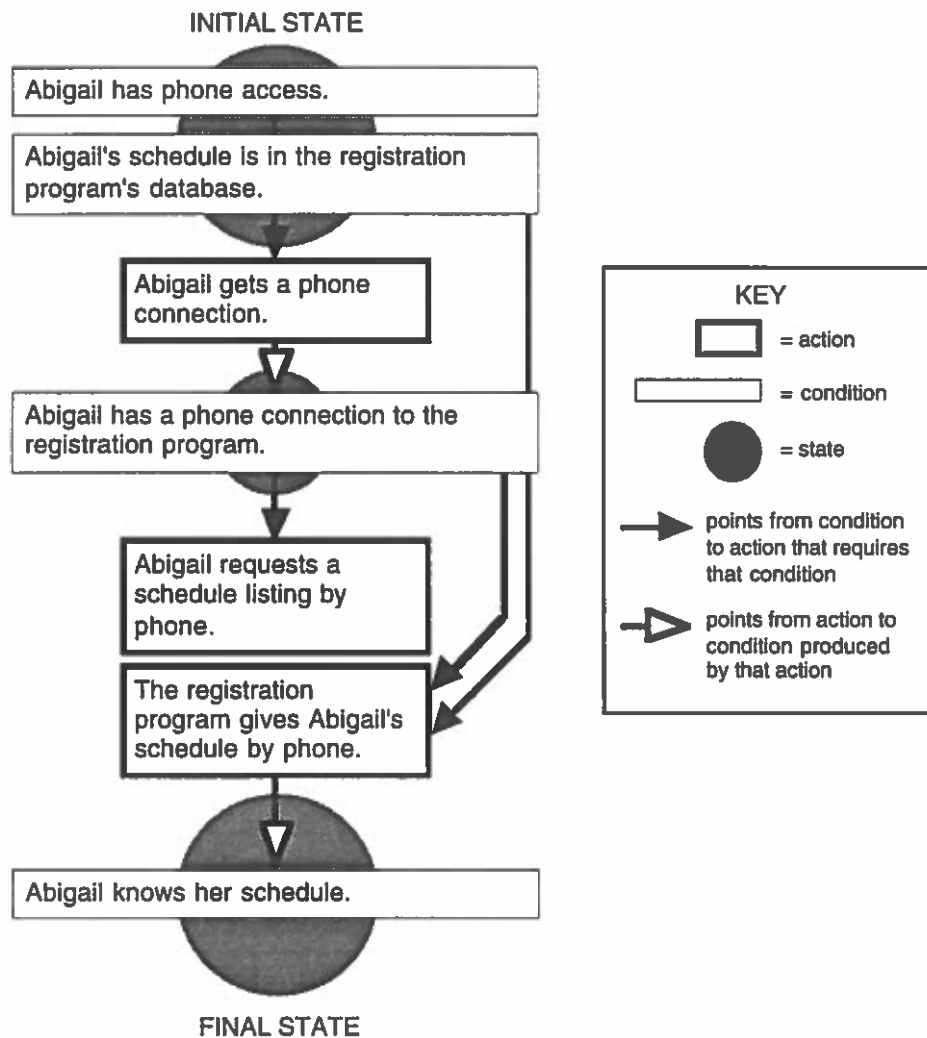


FIGURE 11. A scenario. Not all conditions are shown, and one intermediate state is not shown. The actions and conditions are described with informal text rather than the formal descriptions that GIRAFFE uses.

In Chapter V I describe the representation of scenarios in more detail and explain GIRAFFE's method for finding scenarios. In the next section I give an overview of the

process that GIRAFFE uses to transform requirements and describe the role that scenarios play in that process.

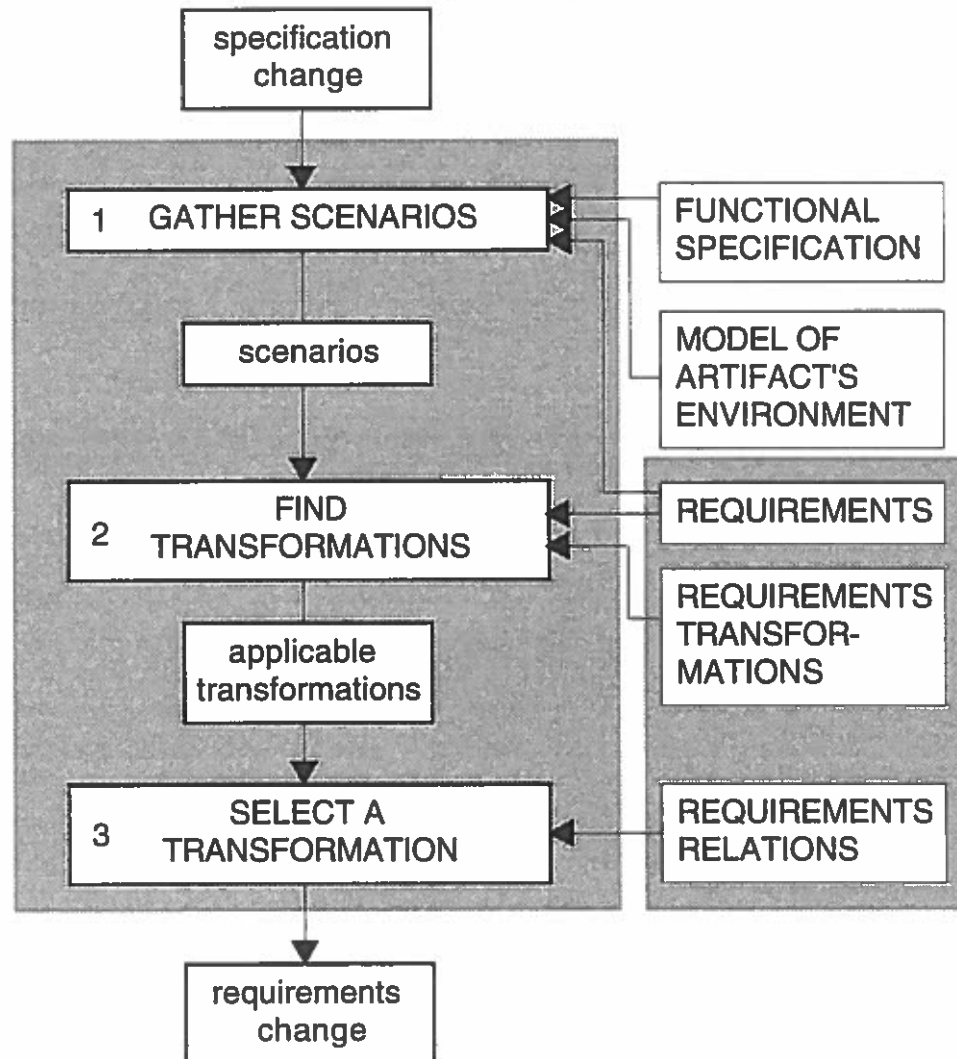


FIGURE 12. The process that GIRAFFE uses to transform requirements.

### The Process

Figure 12 shows GIRAFFE's process for changing requirements. GIRAFFE takes a specification change as input. In the first step of the process, GIRAFFE uses the specification change, along with the rest of the specification, the model of the artifact's

environment and the requirements, to find relevant scenarios showing how agents in the environment interact with the artifact. GIRAFFE uses the requirements to specify initial and final states for the planner, OPIE, to use in finding scenarios. GIRAFFE uses the specification and the model of the environment to define the capabilities or actions that can be used in the scenarios.

Suppose that a specification change is to add an action {phone list schedule}. In other words, students can now call the on-line registration system to find out their schedule. GIRAFFE finds various scenarios where students find out their schedules using the new action. Those scenarios are relevant for the requirement <find out schedule>. It also finds scenarios that are relevant to other requirements, such as scenarios where intruders find out students' schedules using the new capability.

In the second step of the process, GIRAFFE uses the scenarios found in Step 1 to find relevant transformations. It uses applicability conditions of the transformations to perform this step. An example of an applicability condition is new scenarios. Chapter IV discusses applicability conditions of transformations.

In the <find out schedule> example, GIRAFFE finds a transformation that changes a path constraint from one where a student must go to the registrar's office to one where the student can go to the registrar's office or get a phone connection. It also finds a transformation for the initial state. In the same step, GIRAFFE finds transformations to other requirements, such as the safety requirement, <schedule privacy>.

In Step 3, GIRAFFE lists all of the transformations found with their ratings and lets the client select the transformations to apply. The ratings are based on GIRAFFE's knowledge of requirements relations. They indicate the program's conclusions about the strongest requirements satisfied by the current specification (after the change). In the <find out schedule> example, GIRAFFE's rating functions tell it that a requirement where there

is more than one way for a student to find out her schedule is preferable to a requirement where there is only one way, all else being equal.

The two main knowledge components of this process are the requirements transformation library and the rating functions. The transformation library describes the types of changes that are possible by stating the effects of the transformations. It also specifies applicability conditions for each transformation. In Chapter IV I describe in more detail the transformation library and the rating functions. Appendix C gives examples from GIRAFFE's output which show application of the transformation rules and rating functions in GIRAFFE's knowledge base.

### Related Work

In this section I describe work in requirements engineering that relates to the GIRAFFE system. The basic problem of requirements engineering is to create a specification that satisfies the client's requirements. The usual approach is to use the requirements to specify the artifact and then reconcile the specification to the requirements by changing the specification as necessary. In this section I discuss several programs that support this approach, by finding differences between the specification and the requirements, changing the specification, or both.

Table 1 shows the systems that I discuss in this section. As I describe each system, I discuss the system's role in the requirements engineering process and its limitations in addressing intertwining. To facilitate comparison with GIRAFFE, I give examples from the domain of on-line registration for each system. The on-line registration examples are based on examples given by other authors but are my interpretation and are not authoritative.

## Skate

Skate (Fickas & Nagarajan, 1988) is a program that criticizes specifications. Skate takes as input a set of policies and a specification. The policies are determined by the user marking each policy in Skate's model as important, unimportant or unknown. The specification is an extended form of petri net. Skate's critique of the specification has two parts: for each policy it tells whether the policy is supported or obstructed by specification components, and it shows a scenario that illustrates to the user how the support or obstruction takes place.

In the domain of on-line registration, Skate might have the following policies:

- Students know what classes they are registered for.
- Maintain the privacy of students' schedules.

Given a specification that includes an action for students to list classes (without giving a password) Skate indicates that the specification supports the first policy but not the second. It shows a simulation scenario where one student finds out another student's schedule. Skate also shows a list of components that support the policy. For example, it might list an action where a student gives a password to list her schedule.

Suppose the person using Skate replaces the list class action with the version that requires a password. Whether or not the new specification supports or obstructs the policies is less clear now. A student can find out her schedule but not as easily as she could in the old specification because she must take the time to enter the password and she might forget it. A student's schedule is more difficult to access but if a student tells her password or forgets to log out from a terminal someone else can still find out her schedule. In Skate's model, a component cannot both obstruct and support a single policy, so Skate cannot include this kind of information in its critiques.



TABLE 1. Systems that support requirements engineering

| Name    | Input  | Output  |
|---------|--|---|
| GIRAFFE | Specification = operator set<br>Specification change = operator added or deleted                       | Requirements = transitions (including environmental constraints)                          |
| Skate   | Requirements = nonoperational policies<br>Specification = extended petri net                           | Critique = supported/obstructed policies, simulation scenarios                            |
| OPIE    | Requirements = transitions   | Critique,<br>Specification = operator set   |
| ISAT    | Requirements = formal, concrete scenarios  | Critique,<br>Specification = rule-based model created by ISAT's user                      |
| SBRE    | Issues<br>Simulation control   | Critique,<br>Specification = rule-based model created by SBRE's user                      |
| ARIES   | Specification changes= application of evolution transformations<br>Requirements = validation questions | Specification = formal model of artifact  |
| RA      | Requirements = informal statement of requirements  | Requirements = formal statement of policies,<br>Specification = "functional requirements" |

Skate finds differences between a specification and policies (requirements) so the client could use Skate's critiques to change policies. For instance, the client could change the schedule privacy policy to be unimportant. However, such a change is more drastic than necessary and Skate does not provide the information for less drastic changes to the requirements. For example, it does not point out a weaker requirement where no one finds out a student's schedule unless the student tells her password. Since Skate gives all-or-

nothing critiques instead of stating stronger or weaker requirements, it does not provide the information necessary to support intertwining in requirements engineering.

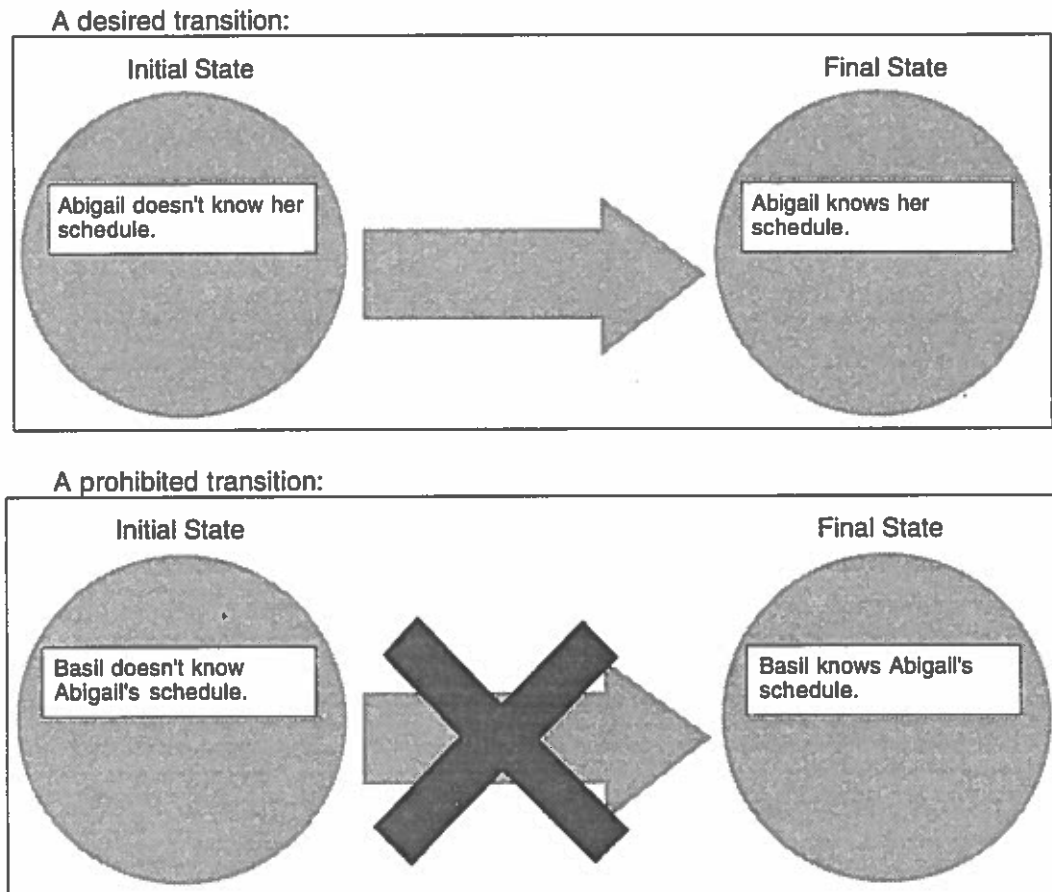


FIGURE 13. Two requirements stated as prohibited or desired transitions.

### OPIE

OPIE (Anderson, 1993) constructs specifications using a process called deficiency-driven specification engineering. In this process, OPIE first acts as a critic to find problems or deficiencies in a specification. Then OPIE proposes changes that address the deficiencies that it has found.

OPIE is given requirements in the form of transitions that should or should not occur in the artifact and its environment. A transition is defined by describing its initial and final states. Figure 13 shows two transitions that might be stated as requirements in the on-line registration domain. OPIE uses AI planning techniques to determine whether or not those transitions can occur.

If a desired transition cannot occur there is an achievement deficiency which OPIE repairs by adding capabilities (planning operators) to the specification. If the desired transition in Figure 13 could not occur then OPIE would add the capability for listing schedules to the functional specification. If a prohibited transition can occur there is a deficiency called a safety violation. Currently repairs of safety violations are not automated in OPIE.

OPIE does not rely on predefined cases as Skate does, and it can automatically find repairs for achievement deficiencies. However, like Skate, OPIE's critiques are all-or-nothing. OPIE has no way of representing partial satisfaction of a requirement.

Suppose that there are two ways for Basil to find out Abigail's schedule: Basil logs in as Abigail (no password required) or Abigail forgets to log out. If OPIE changes the specification so that passwords are required, one scenario is disabled (where Basil logs in as Abigail) but the other remains (where Abigail forgets to log out) so there is still a deficiency. A critique more complete than OPIE's would report an improvement even though one problem still remains.

### SBRE

Scenario-Based Requirements Engineering (SBRE) (Kaufman, Thebaut, & Interrante, 1989) produces a rule-based system that models the proposed artifact. An analyst uses SBRE to find and resolve issues in stating requirements for the artifact. Since the requirements are represented by a rule-based model, they correspond more to the

specifications used by GIRAFFE, Skate and OPIE than they do to the requirements used by those systems and I will refer to the rule-based model as a specification.

SBRE provides automated support for requirements engineering with tools that help the analyst track issues and find scenarios. The process is as follows: The analyst acquires "provisional requirements" during an initial analysis and uses them to create the rule-based model. Next the analyst, with the help of a program called Marcel, finds scenarios which the analyst presents to the client as a means of validating the requirements. The analyst can use Marcel to record issues and positions that arise during validation. In addition, she can link scenarios to positions as a way of representing the justification for a position.

The SBRE process iterates between specification definition to resolve issues and exploration of scenarios to raise new issues. The program does not support intertwining by automatically raising issues when the user changes the rule-based model, nor does the program transform issues in the way that GIRAFFE weakens and strengthens requirements. Furthermore, issues are informally represented and are not a systematic statement of requirements.

### ISAT

Hall (1992) proposed a project called ISAT (Interactive Specification Acquisition Tools) and later (Hall, 1993) describes implementation of one part of the system. In addition to supporting acquisition and validation of a specification, the full ISAT system will also support implementation of a specification and testing of the artifact. Since the validation support feature of ISAT relates most directly to other work described in this section, and since it is more fully described in (Hall, 1993), I will discuss only that part of ISAT here.

ISAT is intended for use with reactive systems. According to Hall, a reactive system is one that responds to external stimuli with relatively few internal state changes between a stimulus and an observable change.

In ISAT, requirements are formal, concrete scenarios created by the client. A specification is a rule-based model of the artifact created by the analyst (software developer). To validate requirements, ISAT simulates the scenario (which is a requirement) with its rule-based model of the artifact and compares the result of the simulation to the observable result stated in the requirement. ISAT also checks for scenario redundancy, describes coverage gaps, and suggests gap-filling scenarios.

ISAT's approach to representing requirements differs from GIRAFFE's in that an ISAT user states requirements as scenarios rather than transitions. In validation, ISAT determines whether the stated actions lead to a desired result (observable condition). In contrast, GIRAFFE determines whether *any* sequence of actions leads to a desired result. Thus in ISAT there is an assumption that the client already knows what actions are relevant to achieve a certain result. For example, a client can state that if a student calls in and requests a schedule listing that she will hear a schedule listing. However, there is no way to state that there has to be *some* way for a student to find out her schedule.

For the same reason, safety requirements are difficult to state in ISAT's requirement language. The client can only specify that a certain state should not occur at a certain point in a scenario. There is no mechanism for requiring a condition to not occur at all. Thus the <schedule privacy> requirement cannot be stated in ISAT's requirement language.

Although ISAT can suggest changes to initial states of requirements based on coverage analysis, the changes still assume the same sequence of actions. Thus there is only a limited opportunity to change requirements, and the changes are not based on alternative paths. There is no way to transform a requirement when a specification change

makes a new path for a transition possible, since requirements are based on a given sequence of actions.

## ARIES

ARIES (Johnson & Feather, 1991; Harris, Johnson, Benner, & Feather, 1992) is a system that supports requirement analysis and specification development. One of the main features of ARIES is its library of evolution transformations. The transformations allow the analyst to explore the possible space because transformations can be undone and replayed. Furthermore, the transformations automate many low-level editing tasks and thus reduce the chance of low-level errors.

Another important feature of ARIES is its ability to represent requirements along several semantic dimensions. The dimensions are modular organization, entity-relationship model, information flow, control flow and state transition descriptions. The dimensions are useful for characterizing the effects of the transformations in the system's library and for allowing the system to describe requirements to the user in various notations.

ARIES criticizes specifications using constraint propagation mechanisms and preconditions of transformations. The preconditions of transformations help prevent the analyst from applying transformations when they are not appropriate, and constraint propagation mechanisms allow more thorough consistency checks.

The ARIES Simulation Component (ASC) helps the analyst find problems in the specification by simulating execution of the artifact. ASC uses simulation to answer validation questions. ASC has the capability of finding the relevant parts of a specification and so can simulate the artifact's behavior with respect to the validation question when other parts of the specification are incomplete. ASC also uses approximations given by the analyst to make simulation possible when the specification is incomplete.

In the domain of on-line registration, a validation question might be "Can a student find out her schedule?" In other words, is <find out schedule> satisfied? ASC uses simulation to find traces of the system to answer the question. It responds "always", "at least once" or "never" depending on how many traces there are. Unlike OPIE, ASC bases its answer on more than one trace (or more than one plan, in OPIE's terminology).

ASC does not address the same problem that GIRAFFE does because it does not suggest changes to requirements. The purpose of ASC is to answer validation questions in the face of an incomplete or inconsistent specification, whereas the purpose of GIRAFFE is to find the strongest requirement satisfied by the specification. ASC takes a requirement (validation question) and a set of assumptions as given and reformulates the specification to find the answer, whereas GIRAFFE takes the specification as given and makes changes to the requirement. Instead of using a given set of assumptions to see whether a requirement is satisfied, as ASC does, GIRAFFE finds the weakest set of assumptions that allow the requirement to be satisfied.

### The Requirements Apprentice

The Requirements Apprentice (RA) is an instantiation of an acquisition program called the Listener developed by Reubenstein (1990). The purpose of the Listener in general and the RA in particular is to use domain knowledge and inference to construct a formal statement from an informal one. The domain knowledge is represented as a cliché library, which the Listener uses to disambiguate, make consistent, and complete a given statement (when possible).

The RA accepts an informal description of a software system and produces a formal description. The formal description includes three parts: needs, the environment and the system. The "needs" section of the RA's output most closely corresponds to the notion of requirements used in this paper.

The schedule privacy requirement discussed in previous examples would be represented as a privacy policy in the needs section. The RA uses cliches to build representations of privacy policies but its ability to reason about them is limited to acquiring information to fill three roles: a restricted action (list schedule), a restricted group (students) and an authorized group (staff of registrar's office).

Cliches in the RA's library can describe needs; in that sense the RA is capable of suggesting policies. However, the emphasis is on formalizing needs stated by the RA's user rather than on deriving requirements. It has no way of analyzing the degree to which a functional specification satisfies a policy, and no way to transform a policy to make it weaker or stronger. Since it cannot evaluate requirement satisfaction it has no way to suggest requirements changes and support intertwining.

#### Discussion

In this section I discuss the differences between GIRAFFE and other systems in representations, use of scenarios, ability to evaluate specification changes and ability to support intertwining and reuse. The systems described in this chapter differ in the representations they use for requirements and specifications. They especially differ in representation of requirements. First I compare the systems' representation for requirements and then their representation of specifications.

Because of the lack of standard terminology I sometimes use a different term for the input or output of a system than the one used in papers describing that system. In particular, papers on SBRE do not refer to its output as a specification, but since the system model it produces is similar in many ways to the specifications produced by the other systems I refer to it as a specification.



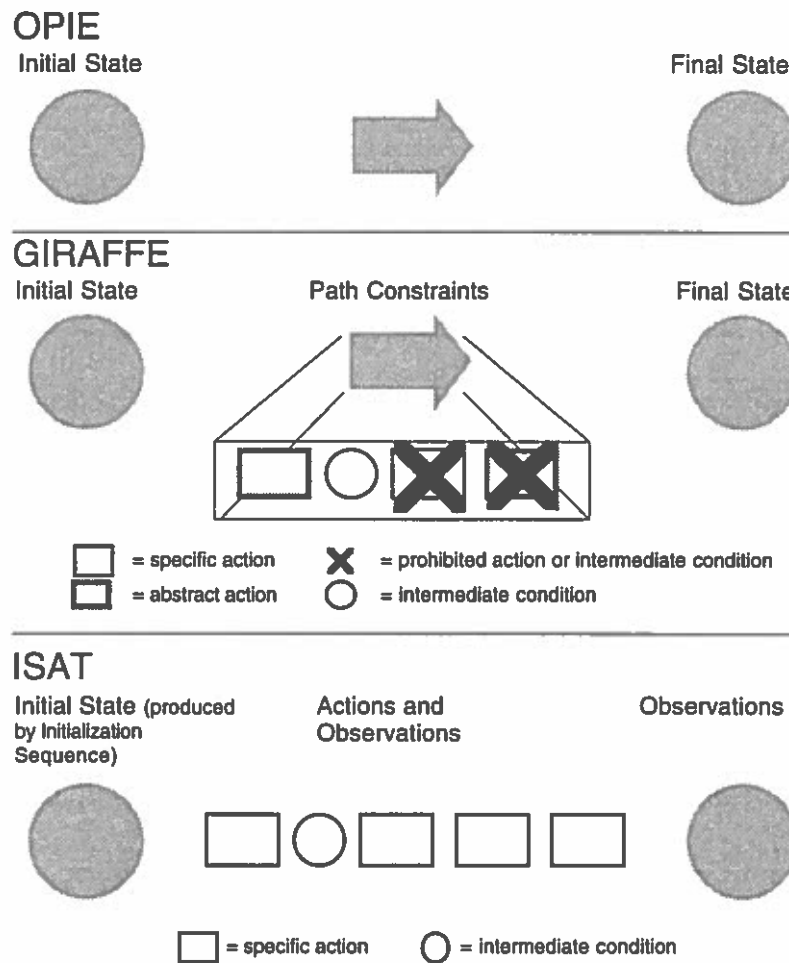


FIGURE 14. Comparison of the representations that OPIE, GIRAFFE and ISAT use for requirements. All three use initial and final states. OPIE has no way of constraining intermediate actions and conditions, and ISAT requires specification of all actions. GIRAFFE represents and suggests constraints on actions and conditions but they are not required. GIRAFFE can also represent constraints on abstract actions which ISAT cannot.

### Representation of Requirements

OPIE, ISAT and GIRAFFE have formal representations for requirements (see Figure 14). OPIE uses transitions with specified initial and final states. ISAT uses scenarios that specify initial and final states but also specify actions and intermediate states. GIRAFFE falls in between. Like OPIE, GIRAFFE uses transitions from initial state to

final state as the basic form of requirements. However, it also includes information about intermediate actions and states, represented as path constraints.

Whereas ISAT uses complete scenarios, including descriptions of actions, GIRAFFE uses transitions that sometimes include descriptions of actions and intermediate states as path constraints but also allows descriptions of abstract actions. GIRAFFE does not use only complete scenarios as ISAT does. Also, ISAT has no notion that corresponds to safety requirements in OPIE and GIRAFFE. Because GIRAFFE's representation can include descriptions of intermediate actions but does not require them, and because it represents safety requirements, GIRAFFE's representation is less restrictive than ISAT's.

Skate uses policies which are uninterpreted text but have links to various specification components. It is difficult to compare representations of requirements for the RA, SBRE and ARIES because they have no formal requirements. ARIES uses text and informal diagram for requirements, although one component of ARIES, ASC, uses a formal representation for requirements that are stated as validation questions.

### Representation of Specifications

The RA is the only system discussed in this chapter that does not produce (or use) an executable model. OPIE and GIRAFFE use the same representation for functional specifications but use them in different ways. OPIE produces a specification whereas GIRAFFE takes a specification (or a specification change) as input. Skate analyzes specifications that are represented as extended petri nets, which is similar to OPIE and GIRAFFE's representation. SBRE and ISAT produce rule-based systems that model the artifact. ARIES produces a model in a representation that subsumes rule-based systems because it can include preconditions and postconditions but can also include additional information such as invariants.

The representations for specifications are similar in many respects, but the representations for requirements are quite different. The similarity in specifications is reflected in the fact that almost all of the systems use scenarios to analyze requirements and specifications. I discuss this use of scenarios in the next section. The difference in requirements is reflected in the fact that the systems' abilities to evaluate specifications and change requirements varies and their ability to change requirements is very limited in most cases.

### Role of Scenarios

The role of scenarios in the different systems varies widely. In Skate, scenarios are used to describe situations to the client and justify Skate's evaluation of whether or not a specification supports or obstructs a policy. OPIE uses scenarios in a similar way but it also uses scenarios to determine what is and isn't possible, and SBRE's use of scenarios is much like OPIE's. ISAT uses scenarios as requirements. ASC uses driving scenarios to model the environment and parts of the artifact that have not yet been specified.

Just as the different systems use scenarios in different ways, they also obtain them from different sources. Some acquire them from a user; others use simulation and planning techniques to find scenarios. In Chapter V I discuss GIRAFFE's method of finding scenarios in detail and compare its method to methods used by other programs.

### Evaluation

I categorize the analysis of specifications performed by the systems as consistency checking, validation and evaluation. Consistency checking looks for inconsistencies within a specification. Validation essentially asks the client "Is this what you meant?". Evaluation tells how well a specification satisfies a requirement or says "This specification satisfies a requirement better than that one."

To evaluate specifications in terms of how well they satisfy requirements, GIRAFFE uses its ability to strengthen and weaken requirements. Skate and OPIE, not having that capability, instead state whether or not a requirement is satisfied (or whether or not a policy is supported, in Skate's case). ISAT can determine how many predicted conditions are actually observed, but its requirement language is more restricted than GIRAFFE's and consequently its ability to find the strongest satisfied requirement is not as great.

The RA and ARIES check specifications for consistency, which is a different issue than evaluating the degree to which a specification satisfies a particular requirement since it is possible for a consistent specification to still not satisfy its requirements. ASC, which is part of ARIES, helps with validation but does not have the notion of stronger or weaker satisfaction of requirements. SBRE, like ASC, primarily analyzes the specification in terms of validation.

#### Support for Intertwining and Reuse

For a system to support intertwining in requirements engineering it must be able to change the requirements of the artifact. Similarly, a system that can change requirements is better able to support reuse of specification components. Rather than requiring that specification components exist for any set of requirements, a system can find components for the specification and then see how much the requirements would have to change to reuse the component.

When a component is added to a specification, GIRAFFE determines how much the requirements must change. Thus it uses a knowledge base of specification components which can be reused for any artifact in the same domain or in a domain that shares specification components. There is no similar facility in SBRE. The user must determine

the consequences of each change, whether it is addition of a new rule or reuse of a rule from a previous system.

Skate and OPIE both use component libraries that support reuse in specification. The validation performed by Skate and OPIE can tell the client whether reusing a specification component will satisfy requirements as they are stated. However, it does not tell the client how well a requirement is satisfied or if it is partially satisfied. The client has no way to choose among components that all satisfy a requirement, or to choose if there is no component that satisfies a requirement.

Hall (1992) proposes capabilities for ISAT to determine the effects of model changes on satisfaction of requirements (successful execution of scenarios), but since the actions in ISAT's requirements are given, this capability supports development of the model (the specification) rather than changes to the requirements. In other words, ISAT's requirements implicitly define a fixed set of actions which includes each action mentioned in a scenario. That set of actions does not change as the specification develops, only the description of the actions in the rule-based model of the system.

ARIES' transformations provide a way to explore the space of specifications because they can be undone and reapplied. However, since requirements are stated informally (text and diagrams) there is no way to transform the requirements. ARIES is able to produce text that describes the artifact but not the requirements that the artifact satisfies.

The RA is capable of changing requirements, but such changes are limited to filling slots in cliches. It has no way of suggesting transformations that strengthen or weaken requirements as GIRAFFE does. Likewise, ISAT can suggest "gap-filling" scenarios to augment the scenarios entered by its client but cannot otherwise strengthen or weaken requirements.

### Summary

Because of intertwining and reuse, there are times when the requirements, and not the specification, should be changed when the two do not agree. As summarized in Table 2, the other requirements engineering systems I have discussed do not provide the information necessary to make those kinds of changes. Therefore, I have created the GIRAFFE system to support intertwining in requirements engineering.

TABLE 2. Summary of the systems described in Chapter II

| Name    | Analysis of Specifications                             | Suggests Requirements Changes                                | Supports Intertwining |
|---------|--|--|-----------------------|
| Skate   | validation   | —  | —                     |
| OPIE    | validation   | —  | —                     |
| ISAT    | validation   | for completeness   | —                     |
| RA      | consistency  | for completeness   | —                     |
| SBRE    | validation   | —  | —                     |
| ARIES   | validation,<br>consistency                             | to resolve inconsistencies                                   | —                     |
| GIRAFFE | evaluation in terms of stronger or weaker requirements | when specification satisfies stronger or weaker requirements | yes                   |

GIRAFFE addresses intertwining by suggesting requirements changes. In the next three chapters I describe the requirements relations it uses, its knowledge base of transformations and rating functions, and its method of finding scenarios.

## CHAPTER III

### REQUIREMENTS RELATIONS

#### Introduction

GIRAFFE's purpose is to support intertwining by finding the strongest set of requirements that is satisfied by a specification. Having defined "requirements" and "specification" in the previous chapter, I now define an "is-stronger-than" relation for requirements that can be used to decide which is the "strongest" set of requirements. I also define other relations between requirements that can be used to derive new requirements. In the last section I compare the relations used in GIRAFFE to those used in other software-engineering and AI research.

#### Definition of a Requirements Relation

In this section I define the IS-STRONGER-THAN relation between requirements. GIRAFFE uses this relation to find the strongest set of requirements satisfied by a specification change. I first define a relation between sets of requirements, called SET-STRONGER-THAN.

Figure 15 defines the SET-STRONGER-THAN relation between sets of requirements. If neither set is stronger than the other then the two sets are incomparable. Two sets of requirements can be incomparable if neither is a subset of the other. They can also be incomparable if S1 has a requirement that is stronger than the corresponding requirement in S2 but has another requirement that is weaker than the corresponding requirement in S2.

If S1 and S2 are sets of requirements then:

- a1. if S2 is a subset of S1 then S1 SET-STRONGER-THAN S2 .
- a2. if every member of S1 IS-STRONGER-THAN or equivalent to a corresponding member of S2 then S1 SET-STRONGER-THAN S2 .

FIGURE 15. Definition of SET-STRONGER-THAN relation between sets of requirements.

The definition of IS-STRONGER-THAN for sets of requirements depends on a definition of IS-STRONGER-THAN for individual requirements, which is given in Figures 16 and 17. The definition for safety requirements is the same as the one for achievement requirements except that R1 and R2 are reversed. Since the definitions are similar, future references will be to a single figure, Figure 16, instead of to both figures.

For achievement requirements R1 and R2:

- b1. R1 IS-STRONGER-THAN R2 if R1 has a higher (better) value for at least one attribute and is stronger than or equivalent to R2 in other respects.
- b2. R1 IS-STRONGER-THAN R2 if the scenarios allowed by R2 are a subset of the scenarios allowed by R1 and is stronger than or equivalent to R2 in other respects.

From the scenario-subset rule, derive:

- b3. R1 IS-STRONGER-THAN R2 if R1 is equivalent to (or stronger than) R2 in every respect except that R1 includes more general objects than R2.
- b4. R1 IS-STRONGER-THAN R2 if R1 is equivalent to (or stronger than) R2 in every respect except that the qualifications on R1 are a subset of the qualifications on R2.
- b5. R1 IS-STRONGER-THAN R2 if R1 is equivalent to (or stronger than) R2 in every respect except that every qualification on R1 is weaker than, or equivalent to, the corresponding qualification on R2.

FIGURE 16. Definition of IS-STRONGER-THAN for achievement requirements.



The basic assumption for the IS-STRONGER-THAN relationship between requirements is that the client wants transitions described in achievement requirements to be able to occur in as many situations as possible, and transitions described in safety requirements to occur in as few situations as possible.

For safety requirements R1 and R2:

b1. R1 IS-STRONGER-THAN R2 if R2 has a higher value for at least one attribute and R1 is stronger than or equivalent to R2 in other respects.

b2. R1 IS-STRONGER-THAN R2 if the scenarios allowed by R1 are a subset of the scenarios allowed by R2, and R1 is stronger than or equivalent to R2 in other respects.

From the scenario-subset rule, derive:

b3. R1 IS-STRONGER-THAN R2 if R1 is equivalent to (or stronger than) R2 in every respect except that R2 includes more general objects than R1.

b4. R1 IS-STRONGER-THAN R2 if R1 is equivalent to (or stronger than) R2 in every respect except that the qualifications on R2 are a subset of the qualifications on R1.

b5. R1 IS-STRONGER-THAN R2 if R1 is equivalent to (or stronger than) R2 in every respect except that every qualification on R2 is weaker than, or equivalent to, the corresponding qualification on R1.

FIGURE 17. Definition of IS-STRONGER-THAN for safety requirements.

Figure 18 shows a relation whose rules are not based on that assumption. This relation can be used when requirements are incomparable with respect to IS-STRONGER-THAN. However, it can also be misleading. Because these rules can be misleading, rather than being inconclusive, I refer to them as heuristics.

**R1 IS-H-STRONGER-THAN R2 if:**  
 h1. There are more scenarios for R1 than for R2 (heuristic version of scenario-subset rule).  
 h2. More actions are performed by motivated agents; artifact agents are assumed to be motivated.  
 h3. Fewer agents are required for scenarios in R1 than in R2.

FIGURE 18. Definition of IS-H-STRONGER-THAN (is-heuristically-stronger-than) for requirements.

In the remainder of this section I formally define the rules given in Figures 15-17. First I define and discuss the rules for sets of requirements which are shown in Figure 15. Next I examine the attribute value and scenario-subset rules, which are the basis for the IS-STRONGER-THAN relation shown in Figure 16. Then I show how the object generality (b3) and generality of qualifications (b4 and b5) rules can be derived from the scenario-subset rule and state the reason why they are more useful than the general scenario-subset rule. Finally I discuss some heuristics for deciding if one requirement is stronger than another.

### Requirement Sets

In this section I discuss how requirement sets might be weakened by removing requirements and in later sections of this chapter I discuss situations where a client might strengthen a set of requirements by adding new requirements which have been derived from other requirements.

From Rule a1 one can conclude that removing a requirement weakens the set. The formal definition of Rules a1 and a2 is:

Definition 3

Let  $S1$  and  $S2$  be sets of requirements. There are two rules for concluding that one set of requirements *is stronger than* another:

(a1) If  $S1 \supset S2$  then SET-STRONGER-THAN( $S1, S2$ ).

(a2) If  $\exists r1 \in S1, \exists r2 \in S2, \text{CORR}(r1, r2) \wedge \text{IS-STRONGER-THAN}(r1, r2)$   
 $\wedge \neg(\exists r3 \in S1, \exists r4 \in S2, \text{CORR}(r3, r4)$   
 $\wedge (\text{IS-STRONGER-THAN}(r3, r4) \vee \text{INCOMPARABLE-REQS}(r3, r4)))$   
 then SET-STRONGER-THAN( $S1, S2$ ). □

Two requirements,  $r1$  and  $r2$ , *correspond* if they have the same name: If  $r1.name = r2.name$  then  $\text{CORR}(r1, r2)$ . Two requirements are *incomparable* if neither is conclusively stronger than the other:

Definition 4

Let  $r1$  and  $r2$  be requirements.

If  $(\text{MORE-SCENARIOS}(r1, r2) \wedge \text{BETTER-ATTRIBUTE}(r2, r1))$   
 $\vee (\text{BETTER-ATTRIBUTE}(r1, r2) \wedge \text{BETTER-ATTRIBUTE}(r2, r1))$   
 then  $\text{INCOMPARABLE-REQS}(r1, r2)$ . □

If a requirement cannot be satisfied, the client might simply abandon it, causing the new requirements set to be a subset of the old one, and thus weaker by Rule a1 of Definition 3. Abandoning a requirement is an all-or-nothing approach to transforming requirements: for example, the client decides that it's impossible to keep schedules private and drops <schedule privacy> completely. A tenet of GIRAFFE research is that abandonment is the last resort (unless a requirement becomes inappropriate, as opposed to infeasible).

A less extreme way to transform a requirement is to discount, or make less important, the requirement. If importance is interpreted as development priority then in

effect the client says, "Do something about privacy if you have time" or "If you can, make a way for students to find out their grades without getting a transcript". This type of transformation still weakens the set of requirements (by Rule a2 of Definition 3) because importance is one attribute of requirements which affects the relationship between two requirements. In the next section I discuss the role of attributes in the IS-STRONGER-THAN relation between requirements.

Rule a2 (of Definition 3) relates the IS-STRONGER-THAN relation for sets of requirements to the IS-STRONGER-THAN relation for individual requirements. The following sections describe the rules that define IS-STRONGER-THAN for individual requirements.

#### The Attribute Rule

A requirement with a lesser (or less preferred) attribute value than another is diminished with respect to that attribute and therefore weaker, as stated in the following formal definition of the attribute rule (Rule b1):

##### Definition 5

Let  $r1$  and  $r2$  be achievement requirements.

If  $\text{BETTER-ATTRIBUTE}(r1, r2) \wedge \neg \text{BETTER-ATTRIBUTE}(r2, r1) \wedge$   
 $\neg \text{MORE-SCENARIOS}(r2, r1) \wedge \neg \text{INCOMPARABLE-SCENARIOS}(r1, r2)$

then  $\text{IS-STRONGER-THAN}(r1, r2)$ . □

##### Definition 6

Let  $r1$  and  $r2$  be requirements.

If  $\exists a1 \in r1.A, \exists a2 \in r2.A, a1.attr\_name = a2.attr\_name$   
 $\wedge \text{RATING}(a1.attr\_value) > \text{RATING}(a2.attr\_value)$

then  $\text{BETTER-ATTRIBUTE}(r1, r2)$ . □

Suppose that there are two versions of <add class> where one version, <add class 1>, requires the student to go to the registrar's office and <add class 2> does not. <add class 1> is diminished with respect to duration because it has a less preferred value for that attribute.

Attributes that can determine the strength of a requirement relative to another version of the same requirement are importance, duration, likelihood and effectiveness. In GIRAFFE's representation of requirements these attributes do not have precise values; they allow the program to distinguish between requirements with significant differences but do not support making fine distinctions.

Because the attributes do not have precise values, GIRAFFE does not try to resolve conflicts when one requirement has a higher value of one attribute and a lower value of another. For instance, if R1 has a higher likelihood of satisfaction but a lower effectiveness than R2, GIRAFFE considers the two requirements incomparable rather than trying to decide if R1's higher likelihood justifies its lower effectiveness value.

A requirement can have other, problem-specific attributes based on attributes of scenarios or conditions that are associated with the requirement. A client might be interested in the number of students who can get access to the system at one time, for instance. If problem-specific attributes are used, GIRAFFE must have rating functions to tell it which values are preferable. GIRAFFE's rating functions are described in the next chapter.

In the next section I discuss the scenario rule which is another way of determining whether one requirement is stronger than another.

### The Scenario Rule

The scenario rule is based on the assumption that a client wants the transition for an achievement requirement to occur in as many situations as possible, subject to the definition

constraints given for the requirement. Each scenario for the requirement represents an occurrence of the transition in a unique situation, so more scenarios mean that the transition can occur in more situations.

Definition 7

Let  $r1$  and  $r2$  be achievement requirements. The *scenario rule* (Rule b2) for determining whether one requirement IS-STRONGER-THAN another is as follows:

If  $\text{MORE-SCENARIOS}(r1, r2) \wedge \neg \text{BETTER-ATTRIBUTE}(r2, r1)$

then  $\text{IS-STRONGER-THAN}(r1, r2)$ .  $\square$

Definition 8

Let  $r1$  and  $r2$  be requirements. If  $(r1.\text{Scenarios} \supset r2.\text{Scenarios}$

$\vee \text{WEAKER-QUALS}(r1, r2) \vee \text{MORE-GENERAL-OTYPS}(r1, r2))$

$\wedge \neg \text{INCOMPARABLE-SCENARIOS}$  then  $\text{MORE-SCENARIOS}(r1, r2)$ .  $\square$

Two scenarios have *incomparable scenarios* if neither has conclusively more scenarios than the other:

Definition 9

Let  $r1$  and  $r2$  be requirements. If  $\text{WEAKER-QUALS}(r1, r2) \wedge \text{MORE-}$

$\text{GENERAL-OTYPS}(r2, r1)$  then  $\text{INCOMPARABLE-SCENARIOS}(r1, r2)$ .  $\square$

The left side of Figure 19 shows an example where one version of the <find out schedule> requirement (R1) allows three scenarios and another version (R2) allows only two scenarios. Since R2's scenarios are a subset of R1's, GIRAFFE can conclude that R1 allows the transition to occur in more situations than R2.

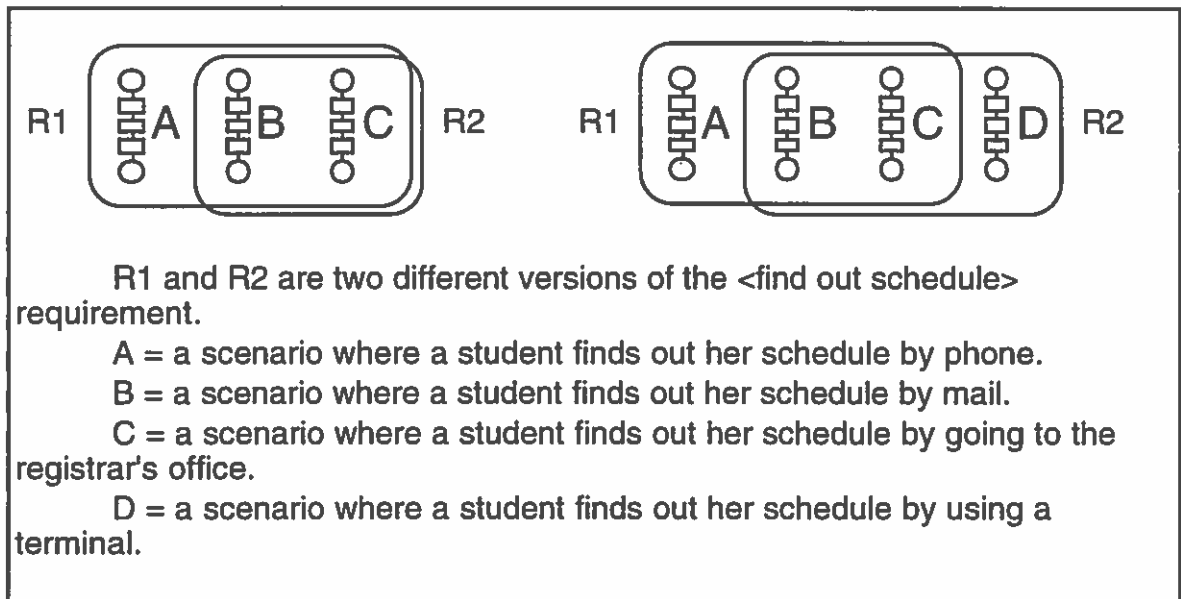


FIGURE 19. Sets of scenarios. The diagram on the left shows a case where R1 IS-STRONGER-THAN R2 by the scenario-subset rule. The scenarios associated with R2 are a subset of the scenarios associated with R1. The diagram on the right shows a case where R1 and R2 are incomparable by the scenario-subset rule because neither requirement's scenarios is a subset of the other's.

The right side of Figure 19 shows an example of incomparable requirements. R1 allows scenarios A, B and C, and R2 allows B, C and D. Neither set of scenarios is a subset of the other and Rule b2 in the IS-STRONGER-THAN definition does not apply.

Rule b2 refers to subsets because otherwise GIRAFFE would have to decide whether two dissimilar scenarios are equally relevant. In the example shown in the left side of Figure 19, GIRAFFE would have to decide between scenarios A and D, which might not be possible given the information available to the program.

If an analyst knows all scenarios possible for the transition of a given requirement, and can easily compare scenarios, then Rule b2 is sufficient. In practice, one cannot expect that kind of omniscience from either a human or a program, so other rules are useful in comparing requirements. The next two sections discuss Rules b3, b4 and b5 which are

consequences of Rule b2 but are easier to apply because they do not require explicit representation of all scenarios.

### Generality of Object Types

Changing the types of objects referred to in a requirement can make the requirement stronger or weaker. The following rules define comparisons between object types:

#### Definition 10

(Rule b3) Let  $r1$  and  $r2$  be requirements.

If  $\text{HAS-MORE-GENERAL-OTYP}(r1, r2)$

$\wedge \neg \text{HAS-MORE-GENERAL-THAN}(r2, r1)$

then  $\text{MORE-GENERAL-OTYPS}(r1, r2)$ . □

#### Definition 11

Let  $r1$  and  $r2$  be requirements.

If  $\exists obj1 \in r1.O, \exists obj2 \in r2.O, obj1.obj\_name = obj2.obj\_name$

$\wedge \text{MORE-GENERAL-THAN}(obj1.obj\_type, obj2.obj\_type)$

then  $\text{HAS-MORE-GENERAL-OTYP}(r1, r2)$ . □

$\text{MORE-GENERAL-THAN}$  is defined in terms of an object hierarchy that is part of the model for a particular domain.

The example in Figure 20 shows two versions of <find out schedule> that are compared based on generality of object types. GIRAFFE compares object types in any part of the transition: initial state, path constraints or final state. To decide that one requirement is stronger than another based on object types, one requirement must be either more general or the same as the other requirement in every respect. GIRAFFE cannot select between two requirements where one is more general in one respect but more specific in another respect than the other.



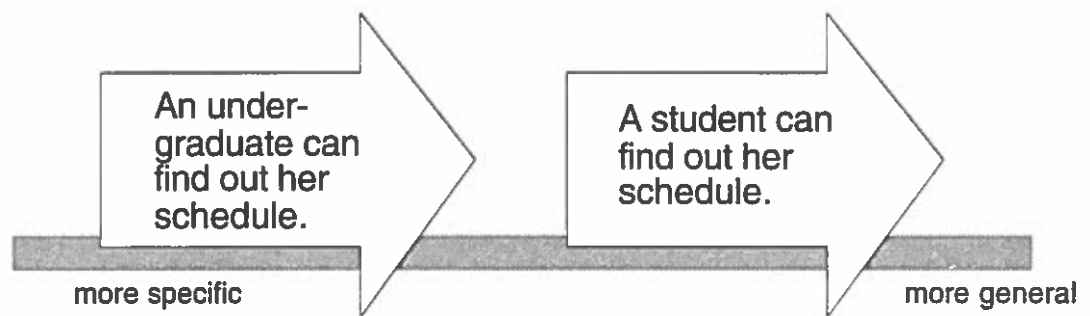


FIGURE 20. Comparison of two transitions based on generality of object types.

The object-generality rule can be derived from the scenario-subset rule if a scenario with a more general object type is considered an abstract representation of all scenarios with that object type's subtypes. For instance, a scenario where a student finds out her schedule represents other scenarios where an undergraduate finds out her schedule and a graduate finds out her schedule. The situation in Figure 20 then becomes an instance of the scenario-subset rule because the scenario for the transition on the left is a subset of the two (or more, depending on how many subtypes of student there are) implicit scenarios for the transition on the right.

An important characteristic of the object-generality rule is that unlike the scenario-subset rule, it does not require an explicit representation of all possible scenarios. Instead it uses a representation of the object generality hierarchy. The object-generality rule is a way of representing and reasoning about classes of scenarios instead of individual scenarios which might be difficult to enumerate.

Two other rules that allow GIRAFFE to reason about classes of scenarios are the qualification rules stated in b4 and b5 of Figure 16. In the next section I discuss those two rules.

### Strength of Qualifications

Rule b4 implies that qualifying a requirement is a way of weakening it, and removing qualifications strengthens requirements. Suppose a requirement says "no student can find out another student's schedule". That requirement can be weakened by qualifying it : "no student can find out another student's schedule unless the student tells her password." If the qualifications from one requirement are a subset of the qualifications for another requirement, then the first has weaker qualifications and is therefore stronger, according to the scenario rule. Comparison of qualifications is formally defined below:

#### Definition 12

Let  $r1$  and  $r2$  be requirements.  $r1$  has *weaker qualifications* than  $r2$  as defined by the rules:

(b4) If  $r2.Q \supset r1.Q$  then WEAKER-QUALS( $r1, r2$ ).

(b5) If HAS-STRONGER-QUAL( $r2, r1$ )  $\wedge$

$\neg$ HAS-STRONGER-QUAL( $r1, r2$ ) then WEAKER-QUALS( $r1, r2$ ) □

In addition to considering subsets of qualifications, one can also reason about the relative generality of individual qualifications. Stronger qualifications restrict the scenarios possible for a requirement more than weaker qualifications. Two aspects of qualifications are important for comparing qualifications: logical implication and the generality of the objects in the qualifications and. Rule 1 in Definition 13 addresses logical implication and Rules 2 and 3 address the generality of object types.

Definition 13

Let  $r1$  and  $r2$  be requirements.

- (1) If  $\exists q1 \in r1.Q, \exists q2 \in r2.Q, q1 \Rightarrow q2$  then HAS-STRONGER-QUAL( $r1, r2$ ).
- (2) If  $\exists q1 \in r1.Q, \exists q2 \in r2.Q, q1.q\_type = q2.q\_type = \text{has\_action} \wedge q1.action\_type\_name = q2.action\_type\_name \wedge \text{MORE-GENERAL-OTYP}(q1.agent\_type\_name, q2.agent\_type\_name)$  then HAS-STRONGER-QUAL( $r1, r2$ ).
- (3) If  $\exists q1 \in r1.Q, \exists q2 \in r2.Q, q1.q\_type = q2.q\_type = \text{has\_init} \wedge q1.condition.relation = q2.condition.relation \wedge (\exists obj1 \in q1.condition.objects, \exists obj2 \in q2.condition.objects, \text{MORE-GENERAL-OTYP}(\text{OTYP}(obj1), \text{OTYP}(obj2)))$  then HAS-STRONGER-QUAL( $r1, r2$ ). □

As an example of Rule 3 in Definition 13, consider a qualification for <find out schedule> that states "a student must have access to a terminal." Changing "a terminal" to "an X terminal" weakens the requirement because scenarios where students use other types of terminals are no longer possible. A stronger qualification leads to a weaker requirement.

Rule 1 of Definition 13 compares complex qualifications (composed of AND and OR expressions) in terms of logical implication. Figure 21 shows an example. The transition on the left has a simple qualification in the initial state: the student must know her password. The transition on the right it contains a disjunction: the student must know her password OR she must know her transcript.

The qualification on the right is weaker because it is a logical implication of the one on the left. The weaker qualification restricts the scenarios less than the one on the left—every scenario possible on the left is possible on the right. By comparing the strengths of

qualifications it is possible to reason about the strength of requirements without an explicit representation of all the scenarios possible for the requirement.

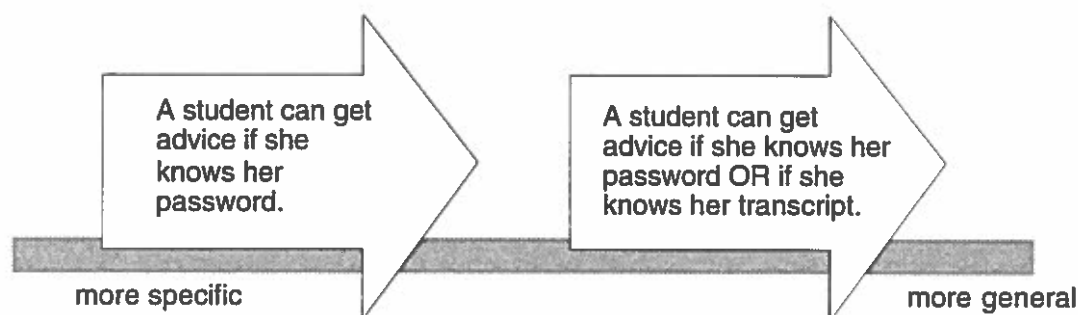


FIGURE 21. Comparison of two transitions based on AND/OR expressions.

GIRAFFE can compare qualifications based on the kinds of conditions used in the qualifications. As an example, consider a requirement that states that a student can register for classes. Suppose that GIRAFFE finds two ways of qualifying this requirement by adding conditions to the initial state. One condition is "the student is in the registrar's office" and the other condition is "phone registration is available." GIRAFFE determines that the artifact has little control over the student's location by analyzing the actions available for the artifact. However, availability of phone registration can be changed by the artifact, and in that sense is a weaker constraint on the initial condition.

- |   |   |
|---|---|
| ■ | static  |
| ■ | changed by actions provided by the environment            |
| ■ | changed by actions that could be provided by the artifact |
| ■ | changed by actions that are provided by the artifact      |

FIGURE 22. Some classes of conditions.

Figure 22 shows several classes of conditions that GIRAFFE considers in analyzing qualifications. In this type of analysis, GIRAFFE might find that a condition is

static with respect to the artifact and the part of its environment modeled by GIRAFFE. Another condition might be produced only by actions provided by the environment. Another possibility is that the condition is produced by an action that the artifact could provide, but doesn't. The condition could also be one that can be produced by the artifact.

Depending on the type of the condition, GIRAFFE considers the qualification stronger or weaker. Conditions that are beyond the control of the artifact are considered stronger qualifications and those that are controlled by the artifact are considered weaker.

### Heuristics

The definition of IS-STRONGER-THAN given earlier provides a way to compare requirements based on attribute values and a single assumption. In this section I discuss some rules that I refer to as heuristic rules because unlike the object-generality rule and the qualification strength rule, they cannot be shown to be correct given the initial assumption that achievement transitions should occur in as many situations as possible. In fact, in some cases they can be misleading rather than being inconclusive.

#### Number of Scenarios

Rule h1 of Figure 18 is useful in cases where the scenario-subset rule does not apply. However, it can be misleading as illustrated by the example in Figure 23. R1 allows two scenarios, A and B which are significantly different. R2 allows three scenarios and so, according to Rule h1, can be considered stronger than R1. However, D and E are very similar to each other, and both are quite similar to C. Even though R2 allows more scenarios than R1, the similarity of the scenarios does not allow the analyst to conclude that R2 is stronger than R1.

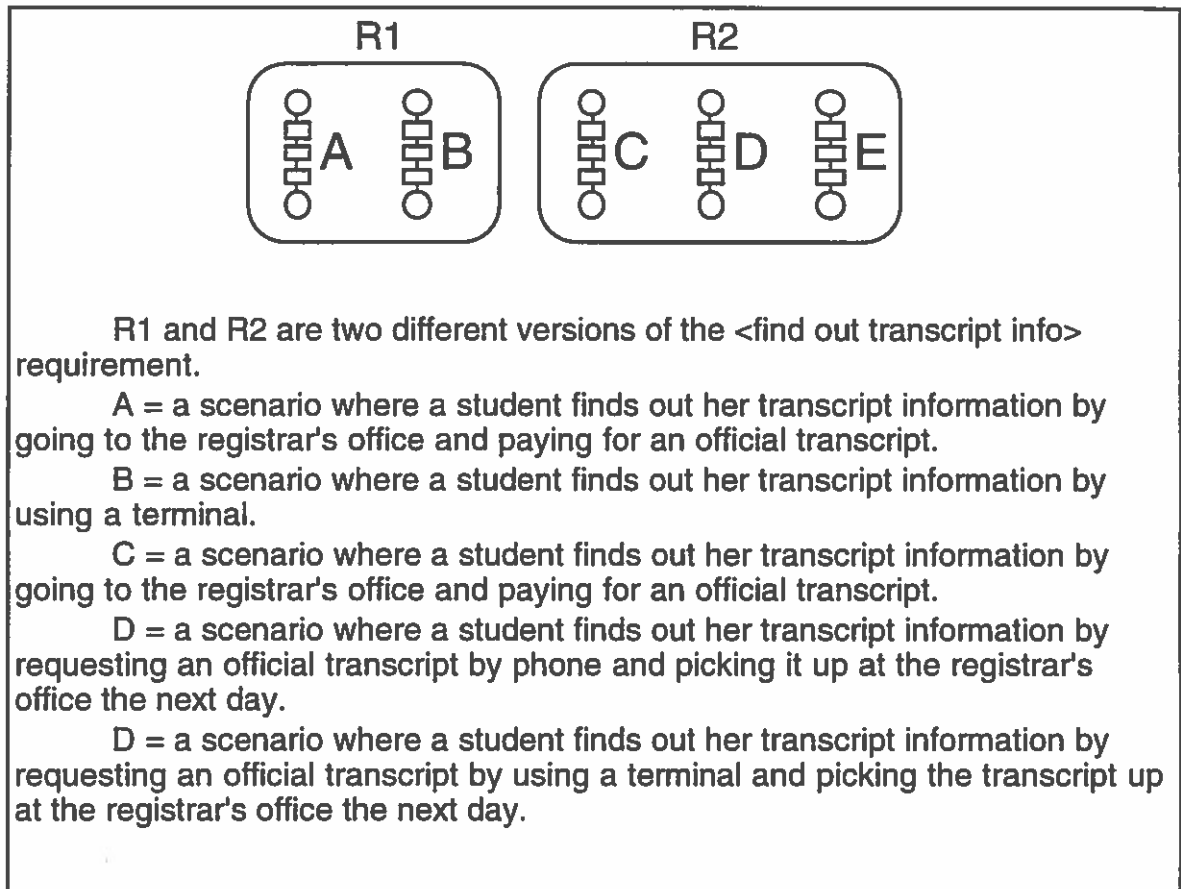


FIGURE 23. A case where the heuristic rule h1 is misleading.

### Types of Agents

GIRAFFE can use knowledge about agent types to compare requirements, as stated in Rules h2 and h3 of Figure 18. Agent types are particularly relevant in analyzing path constraints. Suppose that one version of <schedule privacy> has the path constraint: "no one but the student reads the screen when a student's schedule is displayed." The other version has the constraint: "a student doesn't tell her password to anyone."

In each case, satisfaction of the constraint depends on the actions of an agent that is not part of the artifact. However, in the second case, responsibility rests on an agent who has an interest in satisfaction of the requirement. In the first case, responsibility rests on an

agent with arbitrary motivation. GIRAFFE argues that the second case is preferable because violation of the constraint is less likely. This example is an instance of a heuristic that Dardenne, Fickas, and van Lamsweerde (1991) state in their goal-oriented requirement acquisition method.

To compare requirements based on agent types, GIRAFFE needs domain-dependent relations between types of agents and types of requirements. Note that in the case of <schedule privacy> the relation is part of the domain of privacy rather than being limited to the domain of academic registration.

Rather than comparing requirements based on condition types or agent types, GIRAFFE can base comparisons on domain-specific or problem-specific knowledge about instances of agents and conditions. An example of domain-specific knowledge is comparison of the condition "a student has phone access" to the condition "a student has access to a terminal." The first is more likely to hold than the second. Problem-specific knowledge can be stated by the client. An example of a relation stated by a client (for a particular artifact) is "a student is registered for a class" is a more important final-state condition than "a student knows her grades."

The heuristics discussed in this section are one way of comparing requirements that are incomparable with respect to the scenario-subset and attribute rules. Another approach to dealing with incomparable requirements is to constrain the requirements that are analyzed. Rather than comparing two arbitrary requirements, GIRAFFE compares requirements that are incrementally different and so are less likely to be incomparable. In the next chapter, GIRAFFE's Knowledge Base, I show how GIRAFFE uses specification changes to analyze requirements that incrementally change.

## Summary

In this section I defined the IS-STRONGER-THAN relation between sets of requirements. The definition of that relation refers to the IS-STRONGER-THAN relation between individual requirements, which I also defined.

In defining the IS-STRONGER-THAN relation for individual requirements, I stated a basic assumption on which it is based: a client wants transitions for achievement requirements to hold in as many situations as possible. In other words, there should be as many scenarios as possible for those transitions. Likewise, there should be as few scenarios as possible for transitions of safety requirements. The scenario-subset rule is based directly on that assumption, and the rules for object generality and qualification strength are in turn based on the scenario-subset rule.

When the scenario-subset rules and the attribute rule don't apply other heuristic rules can be used instead. The heuristic rules are related to the number of scenarios possible for a given transition, the number of agents required for a transition, and the types of agents.

In the next section I discuss relations that can hold between a new requirement and the requirement from which it was derived.

### Other Requirements Relations

In addition to helping a client weaken or strengthen requirements, an analyst can suggest new requirements to the client. The analyst can use her domain knowledge and acquired knowledge of the artifact's environment to suggest new requirements. Since the analyst has no authority to change requirements, she can only suggest them and the client must approve them. In this section I describe the relationship those new requirements have to the requirements from which they were derived.



### Plan Support/Obstruction

One type of requirement that can be derived from another requirement is a support requirement. A support requirement establishes initial conditions for the transition of an achievement requirement and thus makes that transition more likely to occur. Figure 24 shows a support requirement and the achievement requirement from which it was derived. The initial condition, "Abigail knows her password," is required for the transition for <find out schedule> to occur. That condition becomes the final condition in the new support requirement.

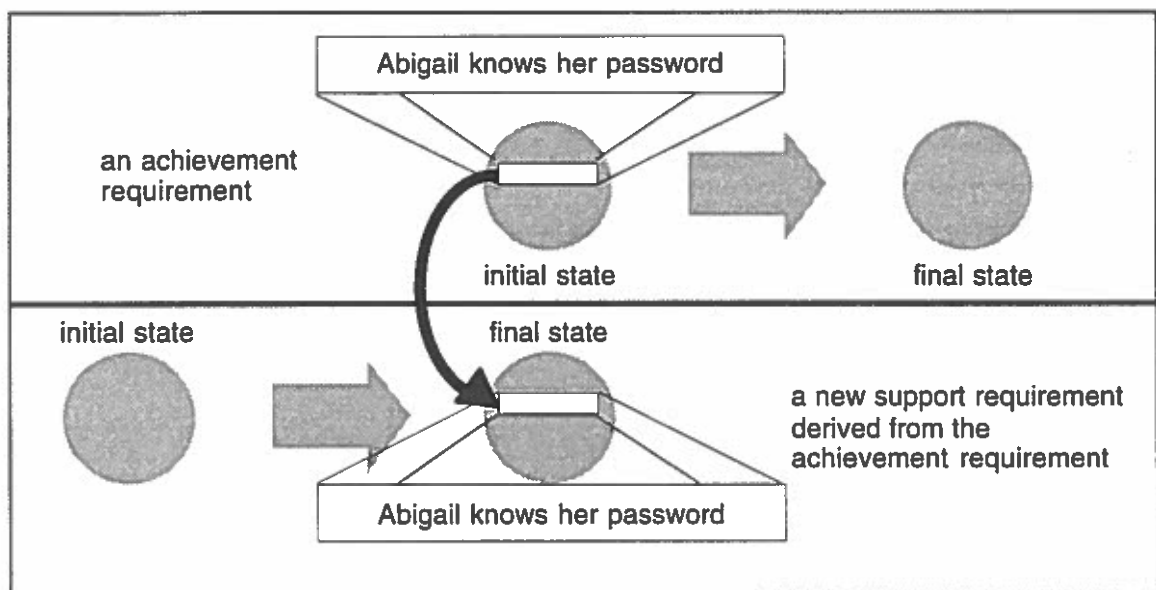


FIGURE 24. A support requirement derived from an achievement requirement.

Just as support requirements make transitions of achievement requirements more likely to occur, obstruction requirements make transitions of safety requirements less likely to occur. In other words, obstruction requirements are intended to reduce the number of safety violations that occur. Figure 25 shows an obstruction requirement and the safety requirement from which it was derived. The safety requirement is <schedule privacy>. An

initial condition in the transition for <schedule privacy> becomes a final condition of the new obstruction requirement.

If an intruder is attempting to find out a student's schedule, one plan the intruder can use is to log in as the student and display the schedule. This plan requires the intruder to know the student's password. By deriving this precondition, the analyst can derive the obstruction requirement that no one else should be able to find out a student's password. The analyst must also consider, however, the possibility that the condition is necessary for the transition of an achievement requirement.

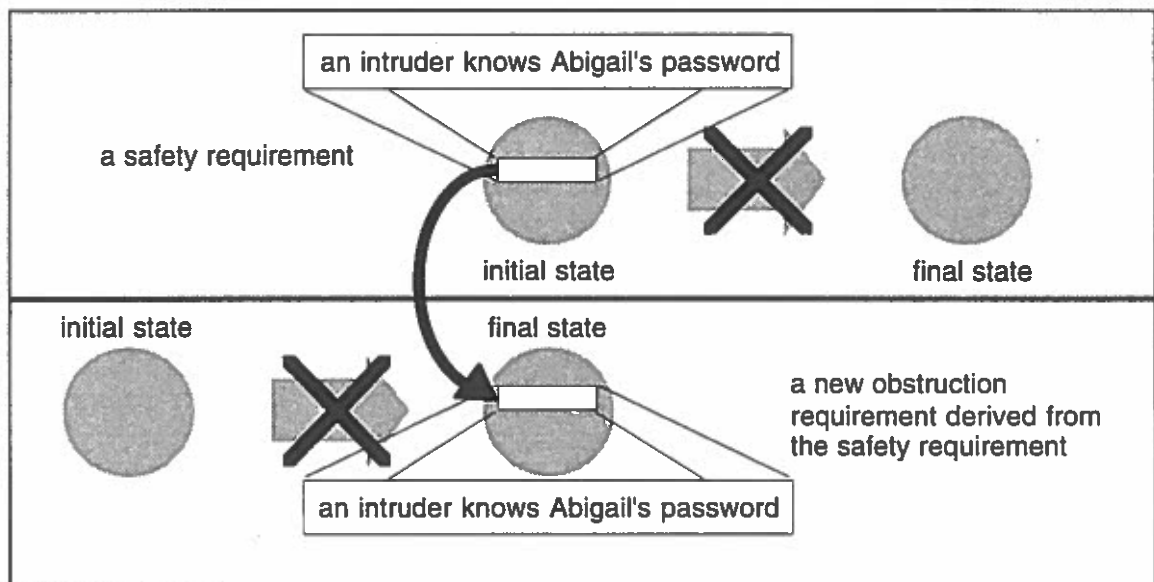


FIGURE 25. An obstruction requirement derived from a safety requirement.

Adding support and obstruction requirements makes a requirement set stronger as defined by the IS-STRONGER-THAN relation for requirement sets defined in the previous section.

## Repair

A client can require an artifact to repair safety violations when it is not feasible to entirely prevent them. For example, <unwanted class> is a safety requirement that says that a student should not have classes that she doesn't want in her schedule. Since it is impossible to prevent a student from adding the wrong class, a client can require the artifact to give a means for undoing an unwanted add.

A requirement to repair a safety violation has a condition from the final state of the safety requirement in its initial state, as shown in Figure 26. The condition "Abigail has CIS121 in her schedule" appears in the final state of <unwanted class> and also appears in the initial state of the new <repair unwanted class> requirement. The final state of the repair requirement includes the negated condition, such as "Abigail doesn't have CIS121 in her schedule".

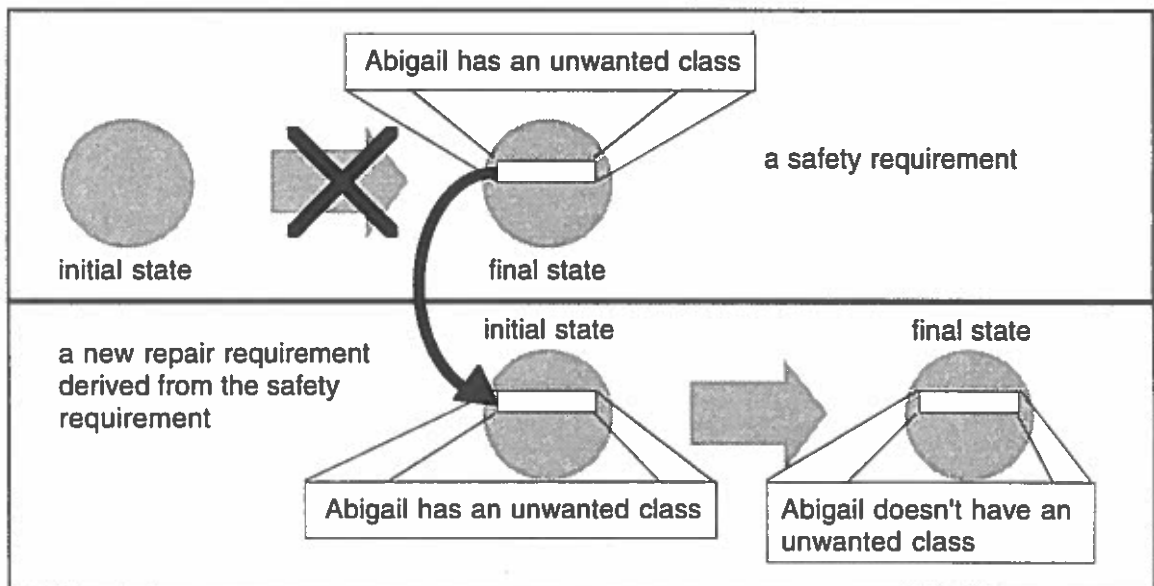


FIGURE 26. A repair requirement derived from a safety requirement.

A repair requirement can replace or augment the safety requirement from which it was derived. If the repair requirement replaces the safety requirement then the requirement is weaker because the repair requirement allows some scenarios prohibited by the safety requirement. A repair requirement that augments a safety requirement makes the requirement stronger if the safety requirement is only partially satisfied.

### Privacy

Several requirements in the on-line registration domain are requirements to give students access to their personal academic information. Examples are <find out schedule> and <find out transcript info>. Privacy requirements are related to that type of personal information requirement. A privacy requirement says that information about a person can only be accessed by that person.

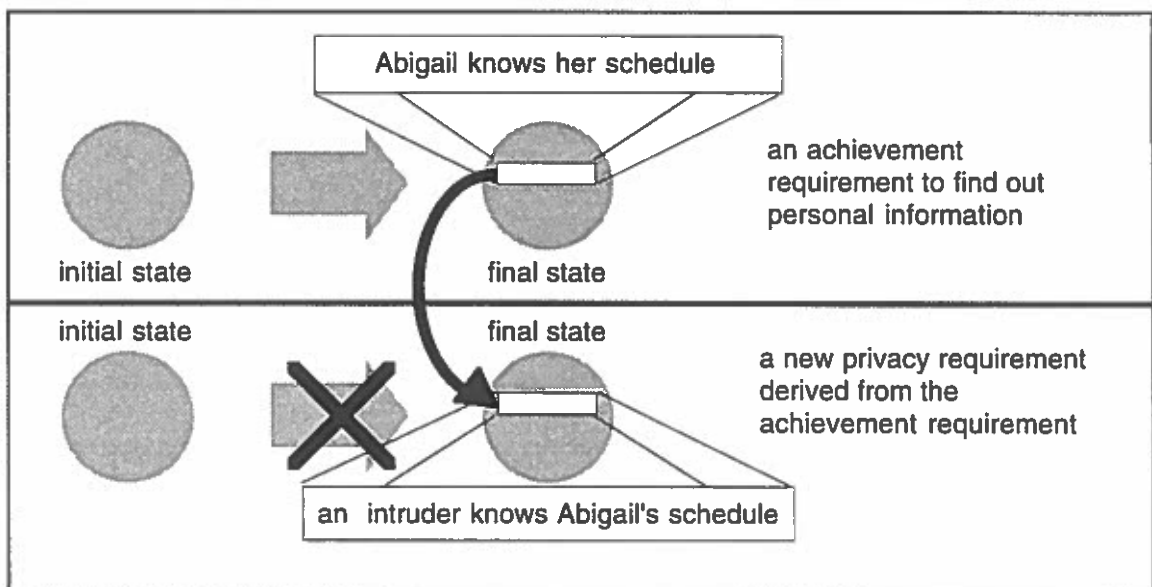


FIGURE 27. A privacy requirement derived from an achievement requirement.

Figure 27 shows how <schedule privacy> can be derived from <find out schedule>. The new privacy requirement in the figure is a safety requirement that says that a student's schedule can only be accessed by that student.

Adding a derived privacy requirement to a requirement set makes the set stronger because the new requirement prohibits some scenarios that were previously allowed.

### Failure

Achievement requirements show how a transition can occur but not how they can go wrong. Failure requirements are safety requirements derived from achievement requirements that show how robust the client expects the artifact to be. For instance, <add class> is an achievement requirement that says the artifact should provide a way for a class to be in a student's schedule. <add class failure> is a safety requirement that says that the add will not fail. The derivation of <add class failure> is illustrated in Figure 28.

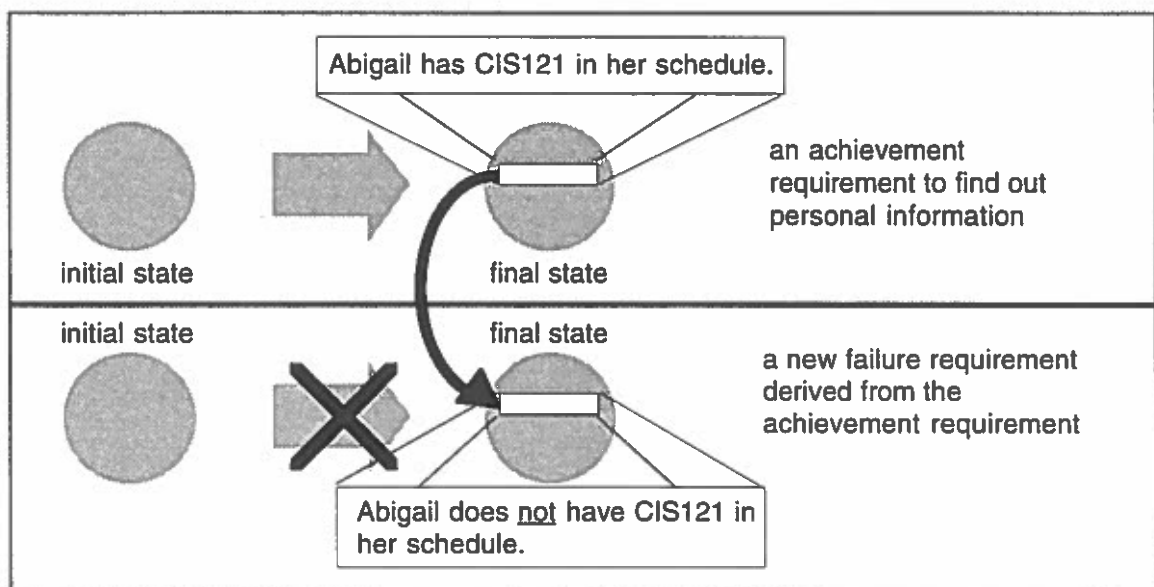


FIGURE 28. A failure requirement derived from an achievement requirement.

Failure requirements in GIRAFFE are not intended to show that the artifact is fail-safe for two reasons. One reason is the inherent limitations of models. Like any model, GIRAFFE's domain model is incomplete and therefore does not allow GIRAFFE to recognize all possible failures. In addition, failure requirements can be qualified or otherwise transformed. Rather than saying that the <add class failure> transition will never occur, a weaker form of the requirement shows what conditions and actions will cause failure. One qualification might be that the student must perform certain actions, like {get phone connection} and {phone request add}. Another qualification might restrict the ways that other agents can act. If another student takes the last available seat while one student is still getting a phone connection then the transition will fail.

Because each failure requirements is closely related to the achievement requirement from which it was derived, satisfaction of the failure requirement affects satisfaction of the achievement requirement. Satisfaction of a stronger <add class failure> requirement means stronger satisfaction of <add class> because there are fewer ways for the transition to fail.

Although GIRAFFE can represent and suggest failure requirements, it doesn't currently have the ability to analyze satisfaction of these types of requirements because its means of finding scenarios is not well-suited for that type of analysis. In Chapter V I discuss GIRAFFE's method of finding scenarios and its strengths and limitations.

### Summary

Relations between requirements are important for deriving new requirements. Derived requirements can support or obstruction plans (scenarios) for other requirements, repair safety violations, prohibit privacy violations and prohibit plan failure. Deriving these new requirements strengthens the requirement set.

### Related Work

In this section I discuss other research work that relates to the requirements relations used by GIRAFFE. First I look at work that is relevant to the IS-STRONGER-THAN relation, and then I look at work that relates to the derived-requirement relations.

#### Weakening/Strengthening Requirements

Some concepts that are relevant to the IS-STRONGER-THAN relation are fault tolerance, graceful degradation, dimension-based analysis and compromise.

#### Fault Tolerance

Weber (1988, 1989) formally defines fault tolerance in terms of fault scenarios. Stated informally, a fault-tolerant artifact's visible behavior in the presence of faults is equivalent to its visible behavior in ideal conditions. An alternative definition to Weber's is that a fault-tolerant artifact's behavior implements its specification even in the presence of faults. The advantage of using Weber's definition is that it allows one to analyze the fault tolerance of an artifact independently of other correctness constraints.

In the on-line registration example, a fault might be the failure of a network connection. If a student can list her schedule even when a network connection fails, then the artifact is tolerant of that fault. Note that the behavior will not remain the same at all levels of detail. For instance, if a computer uses an alternate connection to get the required information its behavior is not the same as if it uses its primary connection. Thus in order to use this concept of fault tolerance one must define what is considered "visible" behavior.

Saying that an artifact should be fault tolerant is one way of saying that it should satisfy a stronger requirement than a similar artifact that is not fault tolerant. Weber's concept of fault tolerance is useful in requirements transformation because it relates

stronger requirements (fault tolerance) to events that can occur in the environment. In terms of the IS-STRONGER-THAN relation, artifacts that are fault tolerant have fewer qualifications and therefore satisfy stronger requirements than artifacts that are less fault tolerant.

Although Weber's research addresses the question of how to specify fault tolerance, it does not address the question of how to change requirements given a specification change that makes an artifact more fault tolerant. For example, if the <find out schedule> requirement doesn't include any notion of fault tolerance, and the client adds a secondary connection for a computer in the artifact, Weber's method provides a way to represent changes in fault tolerance in terms of the network crashing and other events but does not provide a way to detect or verify changes in fault tolerance. A method that supports intertwining would relate changes in the specification to changes in the requirements, which Weber's method does not do.

### Graceful Degradation

Weber (1988, 1989) defines graceful degradation in terms of a tolerance relation, which is an equivalence relation on behaviors. Whereas a fault-tolerant artifact's behavior is the same whether or not faults occur, graceful degradation occurs if its behavior is equivalent with respect to the tolerance relation. Weber gives an example of a tolerance relation where all behaviors with the same sequence of events are equivalent, regardless of the timing of the events.

In the on-line registration example, if a student can find out her schedule, even if it takes longer, the behavior gracefully degrades in the presence of faults. (Rather than completely disregarding time of completion, as the tolerance relation in Weber's example does, the tolerance relation should probably allow more time but not arbitrary amounts.)



Herlihy and Wing (1991) describe another approach to specifying graceful degradation. They define a relaxation lattice that relates sets of constraints to behavior. Figure 29 shows an example of a relaxation lattice that maps satisfaction of constraints on the environment to different behaviors of the artifact. R1 and R2 are descriptions of the behavior of the artifact and C1 and C2 are constraints on the environment. In the on-line registration example, the behavior and environmental constraints might be stated as follows:

R1 = A student can find out her own schedule.

R2 = No student can find out another student's schedule.

C1 = Each student knows her own password.

C2 = No student knows another student's password.

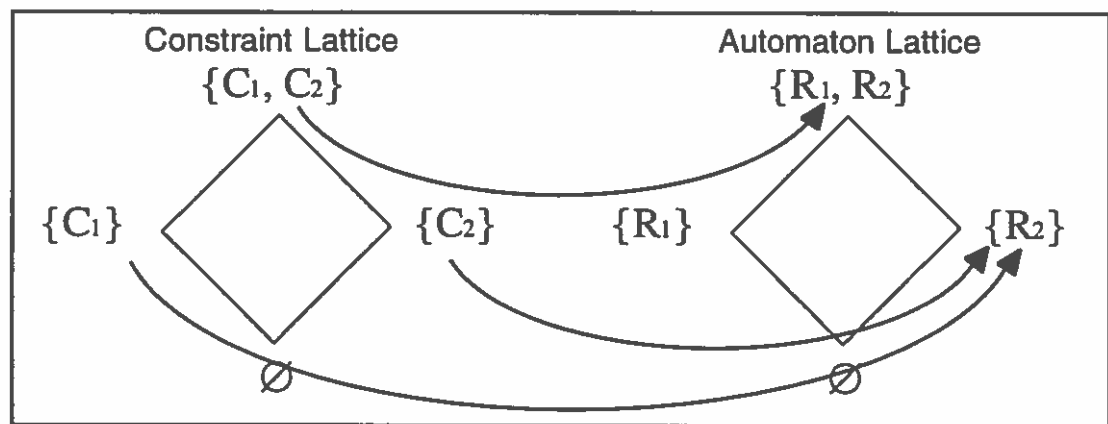


FIGURE 29. A relaxation lattice. Nodes of the Constraint Lattice represents different constraints on the environment, while nodes of the Automaton Lattice represent different behaviors of the artifact. Arcs relate each set of constraints to the behavior specified for that set.

The preferred behavior is {R1,R2} where each student can get her own information but no other student's. As conditions in the environment change, the constraints on the environment will be violated and the artifact will not exhibit the preferred behavior. The relaxation lattice is a way to specify what behaviors are acceptable as conditions deteriorate.

For example, the analyst might specify that if C2 is violated then R2 must still hold although R1 might not. Such a specification might lead to a mechanism where an account is locked when an intruder is suspected; the intruder can't get information but neither can the legitimate user.

Herlihy and Wing's work demonstrates the need for weakening requirements and provides a formalism for representing degradation. However, their only method for weakening requirements is to take subsets of requirements. They provide no way for finer-grain transformation by making individual requirements stronger or weaker. In terms of the requirements relations defined earlier, the automaton lattice represents the requirements subset rule (a1) but not the rules for comparing individual requirements (a2, etc.). In the previous example, the client's only choices are to require R1 (a student can find out her own schedule) or not to require it; there is no way to introduce a weaker version of R1.

Herlihy and Wing's use of a relaxation lattice to specify graceful degradation is similar in some respects to Weber's. In each case, the basic idea is to specify which events will or will not affect the behavior of the artifact. With Weber's tolerance relations, degradation is defined in terms of some formal function, but in the relaxation lattice degraded behavior could be related to behavior that is an arbitrary subset (in terms of constraints satisfied) of the preferred behavior because the client can map each node of the constraint lattice to the automaton lattice independently. One could combine the two methods by defining a tolerance relation for each node of the automaton lattice. If a behavior at a given node was equivalent to the preferred behavior, as specified by the tolerance relation for that node, then the behavior is satisfactory.

### Dimensions

Rissland and Ashley used the concept of dimensions to create a program called HYPO (Rissland & Ashley, 1986) which creates interesting hypothetical cases and argues

about legal cases. In HYPO's domain of common law arguments, the resolution of a case will be the same as the resolution of the previous case that is most similar. HYPO uses dimensions to reason about what "most similar" means.

As an example, Rissland (1986) discusses the use of dimensions in a case involving the warrantless search of a motor home. She shows how the case is strong in two different dimensions which are expectation-of-privacy and inherent-mobility. These dimensions conflict because the laws for searching homes and vehicles are different. Since cases that are strong in two conflicting dimensions, like Rissland's motor home example, are the most interesting hypothetical cases, HYPO has the ability to strengthen and weaken cases along various dimensions to make interesting conflicts.

HYPO strengthens or weakens a case along a dimension by changing values of focal slots. Focal slots can contain numeric, symbolic or boolean values, and HYPO knows how to change those values to make a case stronger along the given dimension. In the motor home case, focal slots might include whether or not the vehicle/home is self-propelled (boolean), how long it has been in the same place (numeric) or the type of setting (symbolic—mobile-home-park versus parking-lot). In HYPO's domain of trade secrets law, focal slots include the number of disclosees (numeric) and the type of information involved (symbolic—technical or vertical). HYPO uses a partial ordering on symbolic values to determine how to strengthen a case.

Fickas and Nagarajan (1988) discuss the use of HYPO-style techniques in generating ranges of examples to criticize specifications. Such examples can be used to motivate either changes to the specification or to the requirements. By making a requirement explicitly address more and more extreme examples, the client is making the requirement stronger (if the artifact is required to address the example) or weaker (if the client specifically states that the artifact does not have to address the example).

One dimension in the on-line registration example might be access to information. A focal slot might be the number of people wanting access at one time, or the amount of time a student has to wait. Focal slots for another dimension, privacy, might include the type of people who find out the information (university employees versus other students) and the type of information disclosed (grades, schedule, major). Although such focal slots represent some aspects of privacy requirements, they do not represent the difficulty or ease that an intruder will encounter.

Wilensky (1983) uses a notion of dimensions to describe partial goal fulfillment in planning. His system represents partial goal fulfillment in terms of scalar values. For instance, if a student wants to find out her grades for the past three quarters, finding out her grades for one quarter constitutes partial fulfillment of her goal. To represent partial fulfillment of privacy requirements in terms of Wilensky's partial goal fulfillment, one would have to identify degrees of privacy which could be represented numerically.

The techniques used by these programs to weaken or strengthen goals along dimensions are useful but they all rely on predefined dimensions, which do not exist for every problem. Furthermore, some requirements, such as schedule privacy, are difficult to state solely in terms of the kinds of dimensions used by these programs.

GIRAFFE uses attributes of requirements to represent numerical and partially-ordered symbolic scales. In addition, it reasons about the relative strengths of requirements in other ways that are not readily represented as predefined dimensions. Its ability to compare qualifications and object generality allow it to determine requirements relations when predetermined dimensions are not available or are not conclusive.

### Compromise

When two parties compromise to resolve a conflict, in effect they each weaken their demands. Robinson (1993) describes ways of detecting and resolving conflicts in a

specification. Robinson's system, Oz, allows development of multiple specifications, each from a different perspective, and uses dimensions and generalization/specialization to find compromises and other conflict resolutions.

As an example of specialization, if one specification allowed students to find out their grades, and another specification did not, Oz might suggest specializing the requirement so that only undergraduates can find out their grades. For a similar requirement with two specifications that conflicted over the number of quarters for which a student could find out grades, Oz could use utility theory to suggest a compromise using the number of quarters as a dimension.

Weakening requirements for compromise and weakening them to support intertwining are related but not identical problems. Compromise uses preferences to select from a set of alternatives which are known to be feasible. Preferences are not an issue in intertwining; instead the issue is what is feasible and how it is stated. The problem in weakening goals to support intertwining is to define the space of feasible solutions at a sufficiently fine level of granularity to state the minimum amount of weakening necessary.

### Summary

Table 3 summarizes the various methods of changing requirements and goals discussed in this section. Weber analyzes fault tolerance in terms of fault events that can occur. Herlihy and Wing relate conditions to subsets of requirements. Each subset of requirements effectively says that requirements not included in the subset are less important. HYPO and Wilensky's program use dimensions and scalar values to analyze similarity and partial goal fulfillment. Robinson's program, Oz, uses dimensions and specialization to find compromises and uses preference relations to choose among various compromises.

In my work on GIRAFFE, I have incorporated these various techniques into a set of requirements relations that allows GIRAFFE to transform requirements and thereby find the strongest requirements satisfied by a specification. Thus GIRAFFE differs from these systems in that it uses all of these types of relations, and it differs in the application of the relations: to transform requirements and support intertwining in requirements engineering.

TABLE 3. Summary of methods for weakening and strengthening requirements

| Name  | Dimensions                                       | Importance/<br>Preference    | Specialize/<br>Generalize         | Conditions/<br>Events                       |
|---|--|------------------------------|-----------------------------------|---|
| Weber—<br>fault<br>tolerance                            |  |                              |                                   | equivalent<br>behavior with<br>fault events |
| Herlihy &<br>Wing—<br>relaxation<br>lattice             |  | subsets                      |                                   | constraints on<br>environment<br>conditions |
| Rissland—<br>dimension-<br>based<br>argument            | numeric or<br>partial<br>ordering on<br>symbolic |                              |                                   |   |
| Wilensky—<br>partial goal<br>fulfillment<br>in planning | scalar values                                    |                              |                                   |   |
| Robinson—<br>conflict<br>resolution                     | used to<br>suggest<br>compromises                | used to select<br>compromise | used to<br>suggest<br>compromises |   |
| GIRAFFE   | attributes                                       | importance<br>attribute      | object<br>generality              | strength of<br>qualifications               |

The work described in this section relates to GIRAFFE's IS-STRONGER-THAN relation. In the next section I describe work that relates to relations between derived requirements.

## Other Requirements Relations

In this section I discuss other work on deriving related requirements and goals. Some concepts relevant to this topic are reuse, decomposition, goal relations and counterplanning.

### Reuse

Reusing requirements is one way to derive requirements for an artifact. The general notion is to draw on experience from other artifacts to state requirements for the current one. The problem is determining which requirements are relevant and how they must be changed to fit the current situation.

The Requirements Apprentice (RA) (Reubenstein) has a set of requirements and specification elements that it reuses. Those elements are called cliches. The RA's approach to reuse is to list for each cliche a set of preconditions that must be satisfied for the cliche to be relevant. By reusing the information in the cliches, the RA acquires a more formal and complete description than its user explicitly stated. The RA's cliches provide more support for choosing a relevant meaning for a policy than for choosing a relevant policy. In other words, the user states a policy and the RA uses its cliches to state the same policy in a more formal way.

As an example, consider <schedule privacy>. By stating that the policy is a privacy policy, the user does not have to explicitly state that the policy involves restricting the agents that can perform a certain action; the cliche gives that information to the RA so that it can choose the relevant meaning. The user can fill roles in the privacy cliche by saying that the action to restrict is {list schedule} and the group that can perform it is a student for herself.

The RA's goal is to decide what the meaning of a stated policy is, rather than to decide when to suggest policies. GIRAFFE suggests changes in policies and thus uses a different approach than the RA. Instead of interpreting a privacy policy stated by the client, GIRAFFE suggests one when appropriate. I describe GIRAFFE's transformations for suggesting requirements changes in Chapter IV, GIRAFFE's Knowledge Base.

The RA uses preconditions to decide which cliches are relevant. Instead of having a program decide what elements are relevant, a requirements reuse system can let the user decide what is relevant. Skate (Fickas & Nagarajan, 1988) takes this approach. It supports reuse by defining a standard set of policies which its user can mark as important, unimportant or unknown. In other words, it helps the user by providing a set of common policies for a domain, but requires the user to explicitly indicate which ones are relevant.

Skate could use information about specification changes to suggest policy changes to the user. For instance, if the user makes a change that introduces a new token type, such as academic scheduling advice, Skate could ask the user whether a new policy, such as <advice privacy>, should be considered important. This kind of analysis would be important for very large domains where the user has many policies and subpolicies to consider, but Skate has no access to the specification editor and so cannot analyze changes in this way.

### Decomposition

The general purpose of decomposition in requirements engineering is "divide and conquer," but there are various interpretations for "divide" (methods of decomposing) and "conquer" (achieving the purpose of decomposition). Depending on the method of division and the purpose of the division, decomposition can be relatively simple or more difficult to automate.



Mylopoulos, Chung, and Nixon (1992) and Nixon (1993) use goal decomposition in dealing with non-functional requirements. Their basic method is to decompose on goal parameters. In other words, the decomposed goals are of the same type (e.g., performance) but have a different parameter. For instance, a performance goal for a system might be split up into performance goals for each structural component of the system. Or, it could be split up for each specialization of a class, such as for each subclass of agent in the system.

Feather (1987) uses decomposition to achieve a design state where each constraint is assigned to individual agents. By decomposing the constraints in this way, the designer can determine what interfaces between agents are necessary. Dardenne et al. (1991) advocate a similar decomposition style as a way of operationalizing constraints.

Whereas the decomposition method described by Mylopoulos et al. is automated, Feather's method is done by hand and Dardenne et al. state that the identification of subgoals is a non-trivial task. One reason for the difference in difficulty of automation is that the method described by Mylopoulos et al. decomposes goals along predefined lines (e.g. a class hierarchy) whereas Feather's method considers a wider range of divisions (e.g. replacing one relation with two others that are not sub-types).

One reason for using decomposition is to factor out similar problems. For instance, if several requirements have scenarios where students begin by getting phone access, the analyst could decompose the scenarios and analyze phone access as a separate problem. GIRAFFE's use of support and obstruction relations serve the same purpose as this kind of serial decomposition. It allows identification and analysis of common problems such as phone access.

### Goal Relations

GIRAFFE's analysis of support and obstruction relations requires the ability to reason about goal relations. Wilensky's research (1983) on metaplanning analyzes goal relations. He discusses various negative and positive goal relations. The negative relations are resource limitations, mutually exclusive states, and conflict with preservation goals. Positive goal relations are goal overlap and goal concord.

Carbonell (1981) uses the term counterplanning to refer to the process of disabling an adversary's plans and overcoming an adversary's obstructions. The basic idea of counterplanning is to find out the adversary's plans and then disable the preconditions of the plan. For instance, by analyzing an intruder's plans for discovering private information, the systems administrator and users can form plans to obstruct the intruder's plans. Disabling preconditions requires a knowledge of goal relations, in particular, mutual exclusion.

### Summary

GIRAFFE uses requirements relations to derive new requirements and to find stronger and weaker versions of requirements. New requirements can support achievement requirements, obstruct safety violations, repair violations of safety requirements, or require privacy of information. Deriving new requirements strengthens the requirement set.

The IS-STRONGER-THAN relation applies to sets of requirements and to individual requirements. The assumption underlying this relation is that the client wants the transition of an achievement requirement to occur in as many situations as possible and the transition of safety requirements to occur in as few situations as possible. The scenario-subset rule, attribute rule, object-generality rules and strength of qualification rules are based on that assumption and define IS-STRONGER-THAN.

Requirements and requirement sets can be incomparable with respect to IS-STRONGER-THAN. One approach to analyzing incomparable requirements is to use heuristic relations. Another approach is to analyze requirements that are similar and thus less likely to be incomparable. GIRAFFE compares requirements after incremental changes to minimize the chances of being unable to compare requirements.

In the next chapter I describe GIRAFFE's knowledge base, which incorporates the requirements relations defined in this chapter.

## CHAPTER IV

### GIRAFFE'S KNOWLEDGE BASE

#### Introduction

Without the requirements relations defined in the previous chapter, GIRAFFE would have no way to know how to change requirements. Using those relations, GIRAFFE can change requirements to find the strongest requirements satisfied by the specification. GIRAFFE's knowledge base incorporates requirements relations into transformation rules and rating functions which allow it to transform requirements and achieve its purpose. In this chapter I describe GIRAFFE's knowledge base.

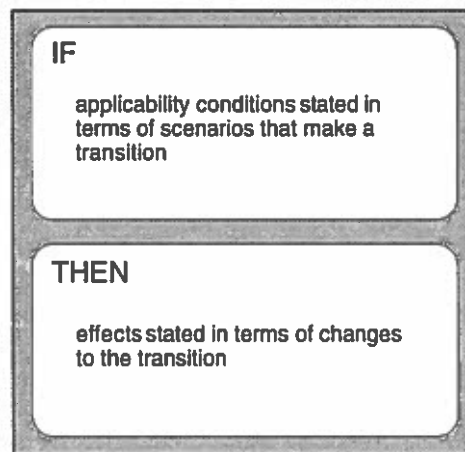


FIGURE 30. Description of a requirements transformation as an IF-THEN rule.

GIRAFFE represents each transformation rule as a set of applicability conditions (or an IF part) and a set of effects (a THEN part) as illustrated in Figure 30. In the next section of this chapter I discuss applicability conditions and how GIRAFFE can tell which transformation rules are relevant. In the third section I describe the various effects that

transformations can have on requirements and in the fourth section I describe the rating functions that GIRAFFE uses to compare transformations.

### Applicability of Transformations

The applicability conditions of a transformation rule are stated primarily in terms of scenarios that exist for a transition and satisfaction of the requirement before the transformation is applied. Some transformations use other information in their applicability conditions such as the kind of specification change being analyzed or the current form of a requirement. A few of GIRAFFE's transformations are meaningless unless the requirement has a disjunction in the initial state, so applicability conditions check for a disjunction to see if the transformation applies to that requirement.

Some of GIRAFFE's transformations use the specification change in their applicability conditions. GIRAFFE can either analyze a complete specification or analyze the changes caused by an incremental specification change. Incremental changes are easier to analyze in some respects because they help GIRAFFE and the client to attribute changes in requirement satisfaction to a change in a specification and they make it easier for GIRAFFE to characterize the change as making the requirement stronger or weaker. Incremental changes also allow some transformations to execute more efficiently because the transformations can use the type of specification change to rule out some requirements changes.

Complex changes are more likely to lead to incomparable requirements than simple ones. Although GIRAFFE can still apply its transformations after such complex changes, it can't characterize the overall change in requirement satisfaction as well as it can for a simple change.

The remainder of this section focuses on the two primary types of applicability conditions: the scenarios that exist for a requirement's transition and whether or not the requirement is satisfied.

### Satisfaction of the Requirement

Since the purpose of GIRAFFE is to find the strongest requirement satisfied by the specification, it will never weaken a requirement that is already satisfied. Therefore a transformation that weakens a requirement that is already satisfied is not relevant. Likewise a transformation that strengthens a requirement that is not satisfied is also not relevant.

For example, one transformation qualifies a requirement by adding conditions to the list of initial conditions required for the transition to occur, thus weakening the requirement. For the <find out schedule> requirement, it might add a condition that the student is at the registrar's office. If the requirement is already satisfied, because there is a way for a student to find out her schedule without being at the registrar's office, then adding the condition is pointless. A transformation that weakens a satisfied requirement is not applicable, as illustrated in Figure 31.

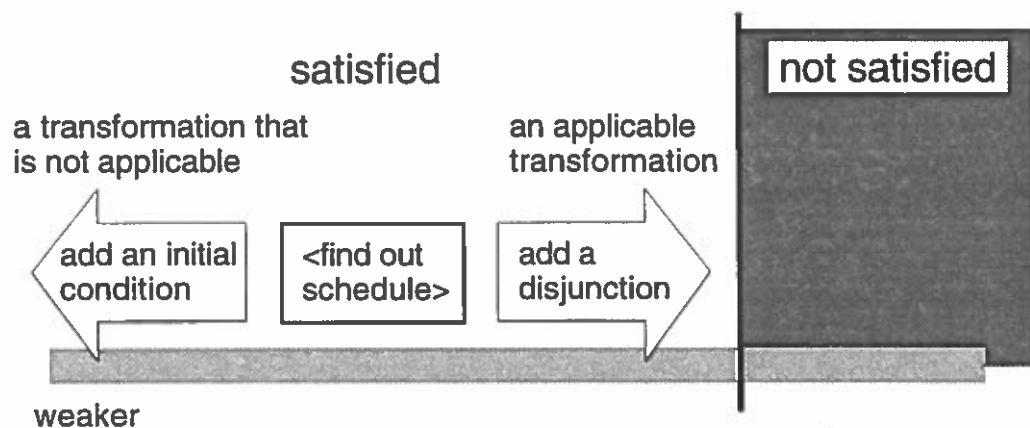


FIGURE 31. Examples of transformations that are applicable and not applicable because of the requirement being already satisfied.

On the other hand, a transformation that strengthens a satisfied requirement is applicable if it yields a stronger requirement that is still satisfied. In the <find out schedule> example, another transformation makes a disjunction in the initial state, making a stronger requirement. That transformation is applicable for requirements that are satisfied and is applicable when the transformation that adds initial conditions is not.

Satisfaction of a requirement, for purposes of applicability, is a function of the type of requirement and the number of scenarios that make the requirement's transition occur. Achievement requirements and repair requirements are satisfied when there is at least one scenario. Safety requirements are satisfied when there are no scenarios possible.

In addition to the number of scenarios possible for a requirement, characteristics of the scenarios are part of applicability conditions. The next section describes how scenarios are used in applicability conditions of transformations.

### Scenarios

Some applicability conditions of a transformation are stated in terms of the scenarios that exist for a transition. Thus, GIRAFFE must know what scenarios are possible for a transition in order to determine whether a transformation is relevant. Figure 32 shows an example of how GIRAFFE uses scenarios in applicability conditions.

The rule on the left side of Figure 32 states that if no scenarios exist for a desired transition (a transition representing an achievement requirement) and scenarios do exist for a similar transition with an additional initial condition, then the initial condition should be added to the transition. The right side of Figure 32 shows the transition before and after application of the rule. The desired transition mentioned in the rule is the transition for the <find out schedule> requirement and the initial condition is "Abigail knows her password."

The application of the transformation shown in Figure 32 is dependent on the functional specification at the time of GIRAFFE's analysis. The same application would

not be relevant for an artifact that did not require passwords for a student to find out her schedule.

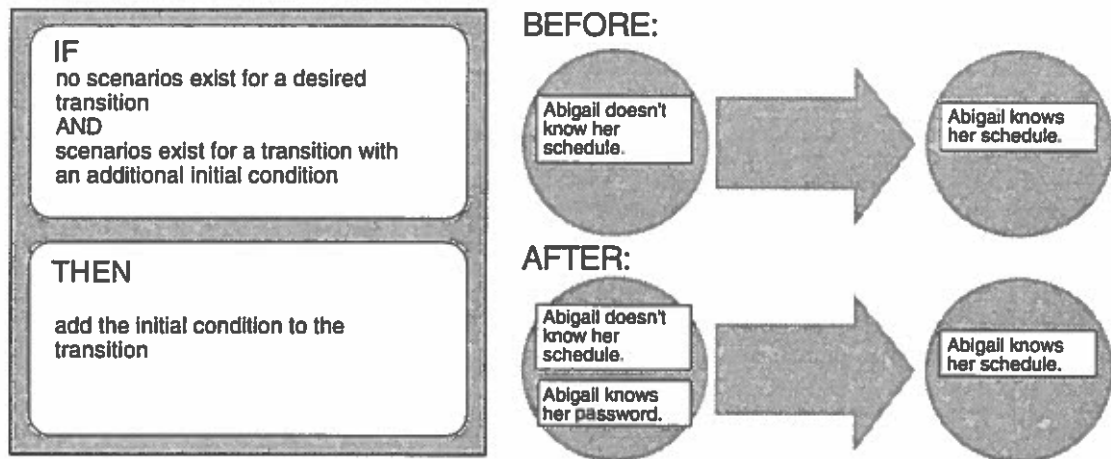


FIGURE 32. A requirements transformation and a sample application.

Figure 33 lists some applicability conditions that can appear in the **IF** part of a requirements transformation. To evaluate applicability conditions, GIRAFFE uses the functional specification and determines what scenarios are possible for a given transition and for similar transitions.

Similar transitions are transitions for weaker or stronger requirements that might have more or fewer initial conditions, different path constraints, or a different final state. Since there are many such transitions, GIRAFFE uses heuristics to confine the analysis to a tractable problem. It uses domain-dependent heuristics to limit the initial states it considers and domain-independent heuristics to limit the length and number of scenarios that it considers. These heuristics are described in the next chapter, Finding Scenarios.



- |  |
|--|
| <ul style="list-style-type: none"> <li>■ whether or not scenarios exist for a given transition</li> <li>■ whether or not scenarios exist for similar transitions with fewer or more initial conditions</li> <li>■ which actions are common to all scenarios for a transition</li> <li>■ which class of agent performs the actions in a scenario</li> </ul> |
|--|

FIGURE 33. Types of scenario applicability conditions that appear in GIRAFFE requirements transformations.

GIRAFFE uses the actions in a scenario to determine when transformations that change path constraints on actions are applicable. In some cases it uses abstract types of actions in scenarios as well. GIRAFFE also uses the class of agent for path constraints since the program only suggests constraints on actions controlled by agents that are not part of the artifact.

The applicability conditions for deriving related requirements differ from those for other transformations. Before deriving a new related requirement, GIRAFFE determines whether there are any actions in the specification, or in the set of all possible actions, that produce the conditions that will be in the final state of the new requirement's transition. As an example, before deriving <repair password privacy>, GIRAFFE determines whether any known actions produce a condition where someone does not know a password. This type of applicability condition is the reason that GIRAFFE derives <repair password privacy> and not a <repair schedule privacy> requirement.

Another kind of applicability condition GIRAFFE uses for deriving related requirements is whether the conditions in the requirement are subtypes of a certain condition type. The `derive_privacy` transformation only derives requirements for achievement requirements that have `knows_info` subtypes in their final conditions and `has_info` subtypes for the same objects in their initial or final conditions.

### Effects of Transformations

There are three basic effects of transformations in GIRAFFE's knowledge base:

- 1) It can qualify a requirement or remove a qualification.
- 2) It can specialize or generalize objects or actions in the requirement.
- 3) It can change the attributes of a requirement to diminish or increase them.

Qualifying the requirement affects the initial state and path constraints of a qualification. Specializing and generalizing objects affects an object wherever it occurs in the requirement so that kind of change can affect the initial state, the final state, the path constraints, or any combination of the three. Diminishing or increasing attributes could affect any part of the requirement as well, but GIRAFFE primarily reasons about path attributes such as duration. Although transformations often cause changes of attributes, GIRAFFE uses rating functions to analyze such changes rather than using transformations to directly change attributes.

GIRAFFE's transformations are designed to produce changes that can be characterized as making a requirement stronger or weaker. Thus the effects of each transformation make changes that correspond to the requirements relations in the previous chapter, rather than making complex changes that are difficult to characterize. Even though each transformation's effects are simple, some changes lead to requirements that are stronger in some ways and weaker in others. Often a transformation will qualify a requirement, making it weaker, but change the duration or some other attribute, making it stronger.

Table 4 (at the end of this section) lists the name and a short description of each transformation in GIRAFFE's knowledge base. The table also relates the transformations to the requirements relations defined in Chapter III.

In this section I first discuss effects on the initial state, then on the path constraints and then on the final state. Finally I discuss deriving new requirements, where all three parts of the transition are affected but the final state is especially important.

### Changing the Initial State

GIRAFFE can change the initial state of a transition by adding or deleting a condition (qualify) or changing the object types of a condition (generalize/specialize). Three examples of these changes appear in Figure 34.

| Type of Change                          | Before Change                     | After Change   |
|---|-----------------------------------|--|
| qualify (add a condition)               | (no conditions)                   | Student's history is in database.                          |
| qualify (make a condition more complex) | Student's history is in database. | Student's history is in database OR Student knows history. |
| specialize                              | Student has access to a terminal. | Student has access to an X terminal.                       |

FIGURE 34. Examples of changing the initial state.

The transformation `X_add_inits` qualifies a requirement by adding conditions to the initial state of the requirement's transition. The transformation's applicability conditions determine when the additional conditions are necessary. For an example of the application of `X_add_inits`, see Appendix C. Another transformation, `X_remove_inits`, removes conditions from the initial state and thus strengthens the requirement by removing a qualification.

In addition to simply adding or deleting conditions, GIRAFFE can form more complex conditions using disjunctions. The transformation `X_add_disj` adds a disjunction to initial conditions and `X_remove_disj` removes one.

As an example, consider a simple version of <find out advised schedule>: "A student should know her advised schedule." This requirement could be represented as a transition with a null initial state and one condition in the final state. GIRAFFE could then qualify the requirement by adding the initial condition: "The student's academic history is in the registration program's database." A more complex, and weaker, version of the qualification is "The student's academic history is in the registration program's database OR The student knows her academic history." (The more complex version might be appropriate when the student can type in the history information necessary for advising.) Appendix C includes a trace where GIRAFFE uses `X_add_disj` to add a disjunction to an initial state.

GIRAFFE distinguishes between initial conditions required for some of the scenarios for a transition and conditions that are required for all scenarios for the transition. Common initial conditions (called CIRPs for Common Initial Required Persistences) do not appear in disjunctions because they are required for all scenarios. When a condition in a disjunction is required by all scenarios, the transformation `X_add_cirp` adds the condition to the list of CIRPs and removes it from the disjunction. The transformation `X_remove_disj` removes the disjunction entirely if all the conditions are required for all scenarios.

### Changing Path Constraints

GIRAFFE can change the path constraints of a transition by adding or deleting a constraint (qualify) or changing the object types of a constraint (generalize/specialize). Transformations indirectly change the value of scalar attributes (diminish). Figure 35 shows examples of these types of changes.

| Type of Change                           | Before Change                                     | After Change   |
|--|---|--|
| qualify (add a constraint)               | (no constraints)                                  | Student enters history accurately.   |
| qualify (make a constraint more complex) | Student enters history accurately.                | Student enters history accurately OR Student responds to confirmation correctly. |
| generalize                               | A student doesn't read user's password on screen. | A person doesn't read user's password on screen.                                 |
| diminish                                 | Adding a class requires 5 time units.             | Adding a class requires 3 time units.  |

FIGURE 35. Examples of changing path constraints.

GIRAFFE treats path constraints for achievement and safety violation scenarios somewhat differently. In both cases GIRAFFE minimizes the number of path constraints. In the case of achievement scenarios GIRAFFE looks for constraints that will assure successful execution of the scenario, while in the case of safety violation scenarios it looks for constraints that will prevent execution of the scenario and thus increase satisfaction of the safety requirement.

Path constraints for achievement scenarios represent assumptions about what actions an agent in the environment will execute. GIRAFFE ignores all scenarios which include actions performed by agents in the environment unless there is an explicit assumption that an agent will perform the action.

For example, for the requirement <add class>, GIRAFFE ignores scenarios with the action {request add} unless there is a path constraint that assumes an agent will perform the action. On the other hand, GIRAFFE does not require explicit assumptions about actions performed by agents controlled by the artifact. GIRAFFE does not require a path constraint for the action {add class} which is performed by an agent controlled by the artifact. The transformations `X_strengthen_ea_pc` and `X_add_etyp_to_ea` add

constraints on actions to achievement requirements. For an example of the application of `X_strengthen_ea_pc`, see Appendix C.

In some cases an agent is not required to execute one particular action for a transition to occur but must execute one of a class of actions. For instance, a student might be required to request an add either by phone or in person. GIRAFFE characterizes these requirements by using abstract actions that are more general than the actions used in scenarios. In this way it can indicate that an agent must execute one of a set of related actions for the transition to occur.

Path constraints for safety violation scenarios represent assumptions about what actions an agent will *not* execute. For example, for the requirement `<schedule privacy>`, GIRAFFE derives a path constraint that a student must not tell her password. If GIRAFFE finds more than one action that must execute for a violation to occur, it creates an AND path constraint. For the safety requirement `<unwanted class>` to be violated when there is a required confirmation, both `{requests wrong class}` and `{confirm wrong class}` must occur. The transformations `X_add_pc` and `X_add_and_pc` add constraints on actions to safety requirements.

GIRAFFE also recognizes constraints on abstract actions for safety requirements. The transformation `X_add_abs_pc` might add a constraint to `<password privacy>`, for example, that no violation will occur as long as an intruder does not guess a password. Since GIRAFFE's domain model includes more than one action for guessing passwords, constraints on abstract actions allow the constraint to be more general.

Fault tolerance (Weber, 1988, 1989) can be analyzed in terms of path constraints. For example, a requirement can state that students should be able to get advice even if a network connection goes down. A transition with a path constraint "Advice system loses network connection" defines the fault scenario. The constraint is not expected to be true, and GIRAFFE ignores (for purposes of that requirement) scenarios where it is not true.

### Changing the Final State

Although there are various possible ways to transform final states of requirements, GIRAFFE has only one transformation that changes the final state. That transformation changes the generality of an object type in a requirement. For instance, it could be that a system originally designed for undergraduates can also be used by graduate students. Then GIRAFFE would suggest changing the object type from `undergraduate` to `student`, as shown in Figure 36. The transformation `x_obj_gen` changes object types in requirements.

| Type of Change | Before Change                | After Change           |
|----------------|------------------------------|------------------------|
| generalize     | undergraduate knows schedule | student knows schedule |

FIGURE 36. An example of changing the final state.

### Deriving Related Requirements

GIRAFFE has transformations to derive five kinds of requirements: privacy requirements, plan support and obstruction requirements, failure requirements, and repair requirements. When deriving a new requirement, GIRAFFE creates a new initial state, final state and path constraints.

To derive a privacy requirement, GIRAFFE copies a requirement that contains a `knows` condition in its final state. GIRAFFE does not create privacy requirements for every achievement requirement that has a `knows` condition in its final state, however. It only creates privacy requirements when the known information is part of a `has_info` relation for a `person`. For example, it will create a privacy requirement for a person's schedule but not for information about a course.

GIRAFFE changes the type of the new requirement from achievement to safety and changes the object from the student who has the information to some other person. For example, <knows schedule> has a condition that a student knows her schedule and is an achievement requirement. <schedule privacy>, derived from <knows schedule>, has a condition that an intruder (an arbitrary person) knows a student's schedule.

TABLE 4. Transformation rules in GIRAFFE's knowledge base

| Name                 | Based on Req. Relation                             | Description  |
|----------------------|--|--|
| X_add_inits          | b4 qualification subset                            | add conditions to the initial state                                    |
| X_remove_inits       | b4 qualification subset                            | remove conditions from the initial state                               |
| X_add_disj           | b5 qualification strength                          | make a disjunction in the initial state                                |
| X_remove_disj        | b5 qualification strength                          | remove a disjunction from the initial state                            |
| X_add_cirp           | b5 qualification strength                          | adds conditions to the common list and removes them from a disjunction |
| X_strengthen_ea_pc   | b4 qualification subset, b5 qualification strength | weaken the constraints on environmental agents                         |
| X_add_etyp_to_ea     | b4 qualification subset, b5 qualification strength | add an action to an environmental agent constraint                     |
| X_add_pc             | b4 qualification subset                            | add a constraint on an action  |
| X_add_and_pc         | b4 qualification subset, b5 qualification strength | add a constraint on a set of actions                                   |
| X_add_abs_pc         | b4 qualification subset, b5 qualification strength | add a constraint on an abstract (more general) action                  |
| X_obj_gen            | b3 object generality                               | change the generality of an object type                                |
| derive_privacy       | privacy  | derive new privacy requirements  |
| derive_plan_support  | plan support                                       | derive requirements to achieve initial conditions                      |
| derive_plan_obstruct | plan obstruction                                   | derive requirements to prevent initial conditions of safety violations |
| derive_failure       | plan failure                                       | derive requirements to prevent failure of achievement requirements     |
| derive_repair        | repair   | derive repair requirements for safety violations                       |



To derive a repair requirement, GIRAFFE makes a new requirement with a final state that is the negation of the final state of a safety requirement. For instance, the final state of <password privacy> has a condition where an intruder knows a student's password. <change password> has a negated form of that condition where the intruder doesn't know the password. The initial state of the repair requirement is basically the final state of the requirement from which the repair requirement was derived.

To derive a plan-support requirement, GIRAFFE makes an initial condition of an achievement requirement the final condition of a new achievement requirement. Plan obstruction requirements are derived in a similar way by making an initial condition of a safety condition the final condition of a new safety requirement.

GIRAFFE derives failure requirements by negating a condition in the final state of an achievement requirement and making it the final state of a new safety requirement.

### Summary

Table 4 lists the transformation rules in GIRAFFE's knowledge base and briefly describes each one. Each transformation qualifies a requirement (or removes a qualification), specializes or generalizes object types in the requirement, or derives a new requirement. Changes in attributes of requirements are not explicitly made by transformations. Instead they are recognized when GIRAFFE uses the rating functions in its knowledge base to analyze transformations. The next section describes GIRAFFE's rating functions and how it uses them.

### Rating Functions

GIRAFFE's purpose is to find the strongest set of requirements satisfied by a given specification. It uses rating functions to compare requirements according to their strength

and their satisfaction. In this section I describe the rating functions implemented as part of GIRAFFE's knowledge base.

### How GIRAFFE Uses Rating Functions

Some of the requirements relations defined in Chapter III are represented as rating functions in GIRAFFE rather than transformations. Unlike transformations, rating functions do not change requirements. Instead they evaluate changes made by transformations as a way of helping the client choose the transformations to apply.

Because some requirements relations are represented as rating functions instead of transformations, the transformations are more simple. If there were no rating functions each transformation would have to include applicability conditions that reasoned about other transformations, other requirement relations, and other requirements.

|  |   |    |  |                  |   |   |      |
|--|---|----|--|------------------|---|---|------|
| <xf397> add a constraint on {give an id} to <transcript_privacy> |   |    |  |                  |   |   | R# 2 |
| Add a path constraint:   |   |    |  |                  |   |   |      |
| + ACTION {give an id} does not occur.                            |   |    |  |                  |   |   |      |
| Rating Summary (R#): 2   |   |    |  |                  |   |   |      |
| constrained in other req xfs                                     | 1 | 1  |  | motivated agent  | t | 2 |      |
| maximum agents   | 3 | 0  |  | minimum agents   | 1 | 0 |      |
| number of scenarios  | 9 | -1 |  | maximum duration | 5 | 0 |      |
| minimum duration   | 1 | 0  |  |                  |   |   |      |

FIGURE 37. An example of the display produced by rating functions. The functions have been applied to a transformation that adds a constraint on the action {give an id} to the requirement <transcript privacy>.

For instance, suppose that GIRAFFE finds a transformation that adds a constraint on {tell password} to <schedule privacy>. GIRAFFE uses the applicability conditions to decide whether the change makes sense or not, but it uses the rating functions to give the client information about what the best change is. If another transformation leads to a

stronger, satisfied requirement, then the change is not the best one. Likewise, if the action is necessary for other requirements or leads to violation of other requirements, as {tell password} does, the change might not be the best one. To include all that information in the applicability conditions would make them interdependent and unnecessarily complicated. Instead GIRAFFE uses rating functions as a modular way of analyzing and comparing different transformations.

GIRAFFE's rating functions analyze the requirements produced by transformations according to various requirements relations and display a summary of the ratings where each type of rating is given a description and a numeric rating. The total of the numbers is displayed but the individual ratings are also displayed so that the client can weight them according to her own judgment and can combine them with her own domain knowledge. Figure 37 shows an example of some ratings given by GIRAFFE to a transformation instance (referred to as x£397) that adds a constraint on the {give an ID} action to the <transcript privacy> requirement. For additional examples of the ratings that GIRAFFE gives to transformations, see the trace examples in Appendix C.

The requirements relations described in Chapter III that deal with attributes are used as preferences rather than constraints and so are represented as rating functions. GIRAFFE does not change attribute values directly but uses attribute values to help decide among transformations. The reasons for taking this approach are that GIRAFFE's attribute values are not precise and so are given less weight than more exact values would receive, and that GIRAFFE's method of finding scenarios has limited abilities of finding scenarios with attributes that have a certain value.

### Description of GIRAFFE's Rating Functions

GIRAFFE has requirements that evaluate the requirements produced by transformations according to their satisfaction and their strength. To evaluate the strength

of requirements, GIRAFFE uses global and local rating functions, which I define in this section. Table 5 summarizes GIRAFFE's rating functions.

To compare transformations according to satisfaction of requirements, GIRAFFE looks at the number of scenarios that are possible for a requirement. If there is one scenario for an achievement requirement then GIRAFFE considers the requirement satisfied, and if there are no scenarios for a safety requirement then GIRAFFE considers the requirement satisfied. All else being equal, GIRAFFE rates transformations that satisfy a requirement higher than a transformation that does not satisfy the requirement. The rating function based on requirement satisfaction is called *R\_satisfaction*.

To compare requirements according to strength, GIRAFFE's rating functions use the requirements relations discussed in the preceding chapter. The following paragraphs describe how those relations are implemented in the rating functions.

GIRAFFE has two types of rating functions: global and local. Local rating functions evaluate a transformation only in terms of the requirement that it affects. Global rating functions look at all the requirements in a set to evaluate the transformations for each requirement. The distinction between global and local rating functions is important because global rating functions are affected by the order of requirements transformation. When GIRAFFE derives new requirements, such as privacy requirements, it indirectly affects the rating of transformations. For instance, a transformation to put a constraint on the action {tell password} gets a lower rating when there are fewer privacy requirements than it does when there are more privacy requirements. Therefore global rating functions might appear inconsistent and so are distinguished from local rating functions.

GIRAFFE's global rating functions look for common path constraints and for inconsistencies between path constraints of safety requirements and achievement requirements. Common path constraints are rated higher because they are weaker

constraints and therefore give stronger requirements. Common path constraints also lead to simpler requirements.

GIRAFFE considers path constraints inconsistent when a path constraint for an achievement requirement conflicts with a path constraint for a safety requirement. For instance, one way for a student to register for a class would be to tell another student her password and then have that student register for her. To consider that a relevant scenario GIRAFFE would have to add {tell password} to the list of actions performed by agents in the environment. On the other hand, the safety requirement <transcript privacy> can be violated by scenarios that include {tell password} so GIRAFFE would suggest a path constraint that {tell password} should not occur. <add class> requires {tell password} for a certain scenario and <transcript privacy> prohibits it, so the two requirements conflict. GIRAFFE allows the client to leave the conflict unresolved. It points out the conflict and gives lower ratings to the transformations as a way of helping the client to state more consistent requirements. For an example of how the `G_not_etyp_xfs` rating function shows inconsistencies, see Appendix C.

The rating function `R_scenarios` rates transformations on the number of scenarios that are possible after the transformation is applied. Achievement requirements with more scenarios are given higher ratings than those with fewer scenarios. This rating function is based on rule h1 for the IS-H-STRONGER-THAN relation described in the previous chapter.

The rating function `R_number_of_agents` rates transformations based on the number of agents required to achieve a desired state. Achievement requirements with fewer agents are given higher ratings than those with more agents. This rating function is based on rule h3 for the IS-H-STRONGER-THAN relation described in the previous chapter.

Since the number of agents required varies from one scenario to another, and since there is often more than one scenario for a requirement's state transition,

`R_number_of_agents` bases its rating on the minimum number of agents and the maximum number. For example, in `<get school info>` it is possible for a student to buy a catalog and read the information or to have someone tell her the information. If both scenarios are possible (and they are the only scenarios) then the minimum number of agents is 1 and the maximum number is 2. `R_number_of_agents` deducts from the ratings of transformations with the highest maximum number of agents and adds to the ratings of the transformations with the minimum number of agents.

Other rating functions that are based on scenario attributes also use minimum and maximum values. `R_duration` adds to transformations with the lowest minimum duration and deducts from ratings of transformations that have the highest maximum duration.

Some rating functions are based on attribute values assigned to actions in the domain model. `R_effectiveness` and `R_likelihood` are based on effectiveness and likelihood attributes. The likelihood attribute is a heuristic value that describes how likely an action is to succeed. For instance, `{guess easy password}` has a higher likelihood value than `{guess hard password}`, where the first action represents guessing a password that is a dictionary word or based on a student's name and where the second action represents guessing a random password. Most actions have the default likelihood attribute but where a general qualitative statement can be made the action attributes provide a way to represent it. Similarly, most actions have default effectiveness attributes but a few have better or worse values.

TABLE 5. Rating functions in GIRAFFE's knowledge base. Functions whose name begins with 'G' are global rating functions; the others are local rating functions

| Name                 | Based on Req. Relation  | Description   |
|----------------------|---|---|
| G_etyp_xfs           | b5 qualification strength, simplicity                                   | increase rating for common constraints  |
| G_new_ea_conflicts   | b5 qualification strength   | decrease rating for action constraints in achievement requirements that are inconsistent with constraints in safety requirements          |
| G_not_etyp_reqs      | b5 qualification strength   | decrease rating for action constraints in safety requirements that are inconsistent with existing constraints in achievement requirements |
| G_not_etyp_xfs       | b5 qualification strength   | decrease rating for action constraints in safety requirements that are inconsistent with proposed constraints in achievement requirements |
| R_duration           | b1 attribute rule   | award or penalize based on duration of scenarios  |
| R_disj_dur           | b5 qualification strength   | award or penalize based on duration of new scenarios  |
| R_triv_disj          | b5 qualification strength   | penalize trivial disjunctions   |
| R_satisfaction       | a1 requirement subset   | increase ratings of transformations that produce a satisfied requirement.   |
| R_pc_generality      | b5 qualification strength   | give AND path constraints higher ratings  |
| R_ea_etyps           | heuristic version of b4 qualification subset                            | decrease ratings of transformations that require more actions   |
| R_scenarios          | h1 scenario count   | award or penalize based on number of scenarios  |
| R_number_of_agents   | h3 fewer agents   | award or penalize based on number of agents   |
| R_pc_motivated_agent | h2 motivated agents   | increase ratings if agent is motivated  |
| R_effectiveness      | b1 attribute  | award or penalize based on effectiveness of actions in scenarios  |
| R_likelihood         | b1 attribute  | award or penalize based on likelihood of scenarios  |
| R_obj_generality     | b3 object generality  | award or penalize based on generality of object types   |
| R_init_pers          | b5 qualification strength, heuristic version of b4 qualification subset | award or penalize based on number of initial conditions   |

Rating functions can incorporate domain-specific information. The `R_init_pers` function rates transformations according to the number and kind of conditions they require in the initial state. It gives higher ratings to transformations where

required conditions can be produced by actions in the artifact's specification or in the set of possible actions for the domain. It also uses domain-specific information about the likelihood of conditions occurring. For instance, in the academic registration domain, students are more likely to have phone access than terminal access. `R_init_pers` uses that domain-specific information to rate transformations.

### Summary

GIRAFFE uses applicability conditions to decide which transformations are possible. In evaluating applicability conditions, the program considers what scenarios are possible for a requirement's transition and for similar transitions. It also considers whether or not the requirement is satisfied so that it can avoid weakening a requirement that is already satisfied or strengthening a requirement that is not satisfied.

The effects of GIRAFFE's transformations include adding and deleting conditions in the initial state of a transition, adding and deleting path constraints on actions, and generalizing or specializing object types in the transition. In addition to transforming individual requirements, GIRAFFE sometimes derives new requirements based on privacy, plan support and obstruction, failure and repair relations.

GIRAFFE's rating functions incorporate domain-independent requirements relations and domain-specific information. Some rating functions are global in that they look at transformations on other requirements, and some are local because they only compare transformations for one requirement.

GIRAFFE's knowledge base of transformations and rating functions allows it to determine what changes in requirements are possible and evaluate those changes with respect to strength and satisfaction of the resulting requirements.



## CHAPTER V

### FINDING SCENARIOS

#### Introduction

GIRAFFE depends on scenarios to perform its analysis of requirements transformations. In this chapter I describe GIRAFFE's representation of general scenarios and its method for finding them.

GIRAFFE uses MOPIE, a modified version of a planner called OPIE, to find scenarios using AI planning techniques. I show how MOPIE differs from OPIE and other planning programs in the kinds of operators that it uses and the plans it produces.

In the last section I look at other methods for finding scenarios that future versions of GIRAFFE could use to find scenarios that are difficult for it to find using its current method. These methods are forward chaining, acquisition and case-based planning.

#### General Scenarios

GIRAFFE uses scenarios that are a sequence of actions that show how a given final state can be achieved from a given initial state. GIRAFFE uses scenarios that are generalized in several respects: they might include more than one path, and the objects are the most general objects possible.

The general scenario encompasses all known sequences of actions that achieve the desired state transition. Each sequence of actions is referred to as a path, so general scenarios in GIRAFFE have more than one path, as shown in Figure 38. Different paths can include the same actions in a different order or different actions. For example, two paths in a general scenario for <find out schedule> might differ because in one path the

student gets a schedule confirmation in the mail and in the other path (shown in Figure 38) the student uses a phone to hear the schedule information.

The general scenario is described in terms of the most general object types possible. So, for instance, if it is possible for any person to find out a student's address, then a general scenario will be stated in terms of a person finding out, not a student. Since students are a subset of people the general scenario still shows that it is possible for one student to find out another's address but also gives additional information about non-students.

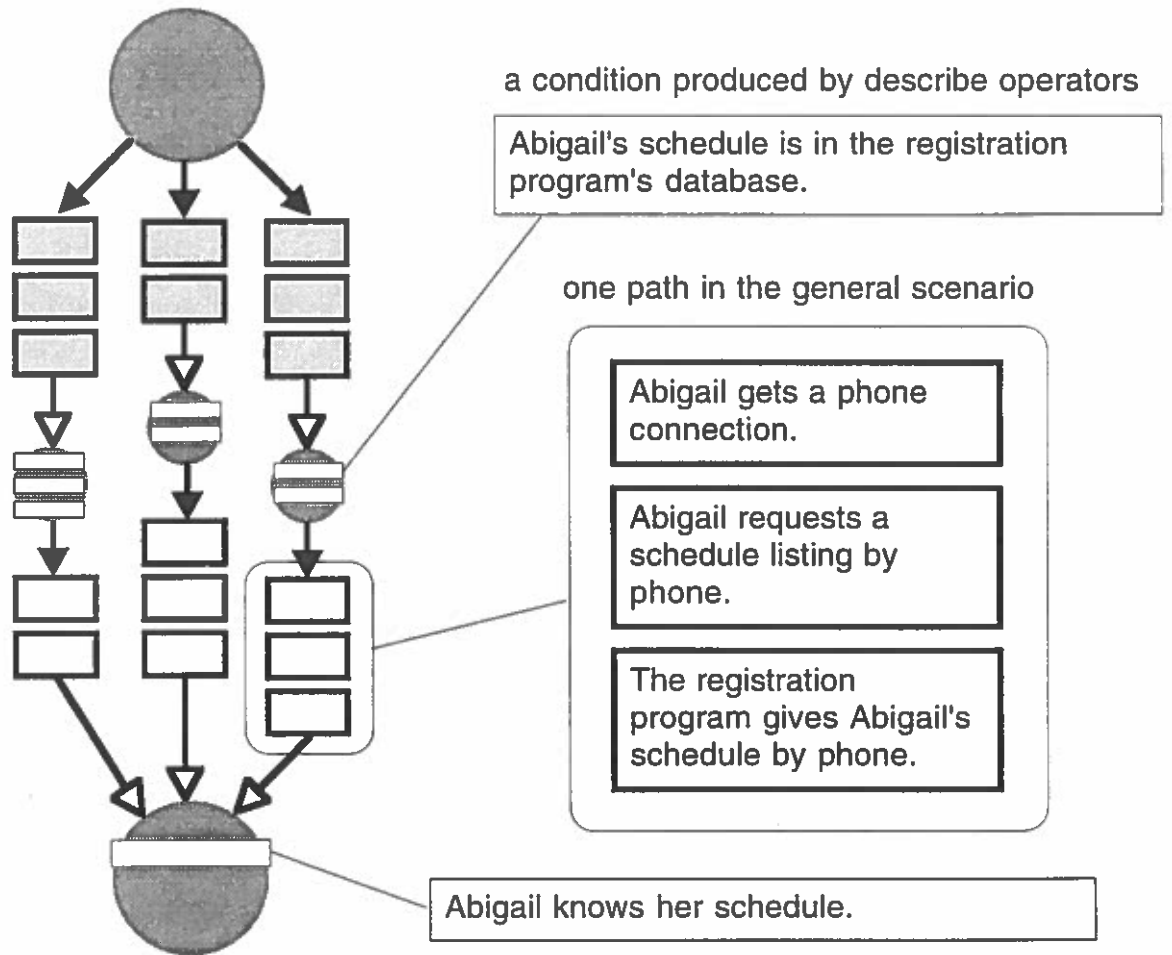
In the next section I describe the planning techniques that GIRAFFE uses to find general scenarios.

#### Using Planning Methods to Find Scenarios

GIRAFFE finds scenarios by describing a requirement as a planning problem and then calling a planner to find plans for that problem. Thus, the issues discussed in this section are how to describe requirements as planning problems, and what planning techniques to use.

The planning program, OPIE (Anderson & Farley, 1988, 1990; Anderson, 1993), uses means-ends analysis to find a partially-ordered set of actions that can achieve the final state of the planning problem from the initial state. GIRAFFE uses a slightly modified version of OPIE to find plans. I will refer to the modified version of OPIE as MOPIE.

initial state of a requirement's transition



final state of a requirement's transition

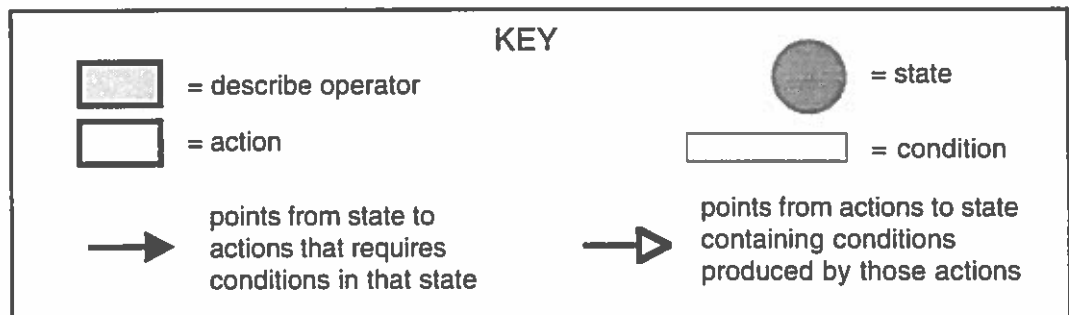


FIGURE 38. A general scenario. The actions in one path of the scenario are shown enlarged. The describe operators produce some conditions used by actions in the path. The state produced by describe operators is considered the initial state of the path and is used to find transformations for the initial state of the requirement.

A planning problem consists of a goal (or final) state and an initial state. A solution to the problem is a plan, or sequence of actions, that produces every condition in the goal state, where every condition required by an action is in the initial state or is produced by an earlier action in the plan.

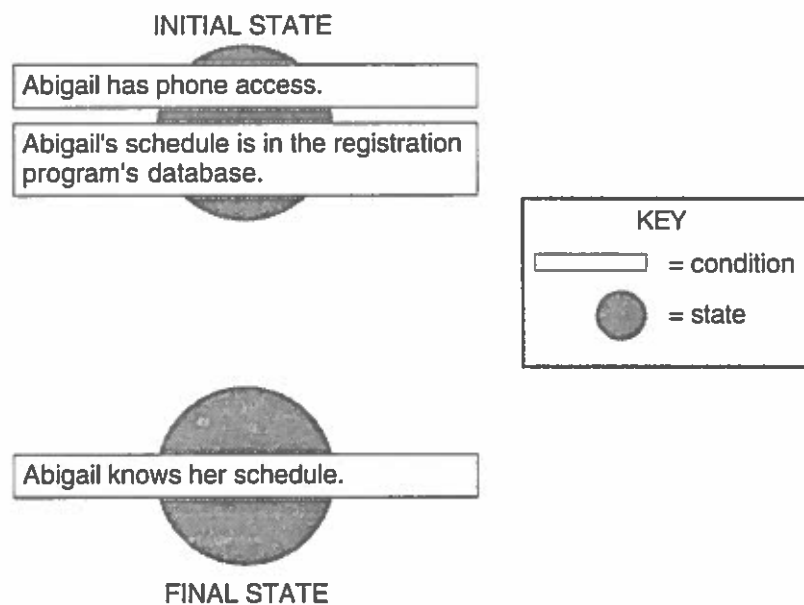


FIGURE 39. A planning problem.

To solve a planning program, OPIE chooses a condition in the final state and checks to see if it is produced by some action. If so, OPIE continues to the next condition in the final state, repeating the process until each condition is produced by some action.

When OPIE finds a condition that is not produced by any action in the plan, it adds a new action to the plan that produces that condition. To choose which action to add, OPIE uses the condition as an index into its hierarchy of actions. It adds the most general action possible to the plan and then specializes the action later as necessary.

For each action added to a plan, OPIE must ensure that the action's preconditions are satisfied. Thus, the preconditions of each action become subproblems that OPIE also

must solve. OPIE solves these subproblems in the same manner as it solves the original problem: by adding actions to the plan that produce necessary conditions.

OPIE treats the initial state as an *initial producer*, a special kind of action that has no preconditions. Without the initial producer, OPIE would usually have an infinite number of subproblems to solve, since most actions have preconditions and thus would lead to further subproblems that could not be solved without adding additional actions. Thus it is important for OPIE to have the initial state specified in the planning problem.

### Describing Planning Problems

As input, OPIE is given a planning problem consisting of a goal state, an initial state, and a set of operators. The operator set specifies the actions that the planner can include in the plan. The operator set that GIRAFFE gives to OPIE comprises the capability set of the specification plus the capabilities of the artifact's environment. The operator set also includes a set of `describe` operators used to define initial states.

In the next two sections I describe how GIRAFFE uses a requirement definition and domain knowledge to define the initial state and final state of a planning program. Although GIRAFFE represents requirements in a form that is basically a planning problem, GIRAFFE must transform the representation in some ways before giving a requirement to the planner as a problem to be solved. Some of the changes are changes to the representation are changes in the problem itself and some are changes that affect the operators used by the planner.

#### Final States

GIRAFFE uses three methods to create final states of plans. It retrieves final states as part of the standard initial requirements stored in the domain model, it derives final states

when deriving new requirements using its related requirements transformations, and it accepts final state descriptions from the analyst.

GIRAFFE's knowledge base of initial requirements provides it with many final states for planning problems. Since GIRAFFE can refine requirements during the course of requirements engineering, requirements can be stored in a very general form and then adapted, with the help of the analyst, to a specific problem. For final states, the adaptation takes the form of modifying object types to make the final state more or less general. For instance, what begins as a requirement for undergraduates to find out their schedules might end up as a requirement for students to find out their schedules.

When GIRAFFE derives related requirements such as privacy, plan-support and repair requirements, it creates new final states. For instance, GIRAFFE might derive a final state of a safety requirement where an intruder knows a student's schedule. The transformations described in Chapter IV provide the information that GIRAFFE needs to derive that kind of final state.

GIRAFFE can accept requirements that describe final states from the analyst, but such requirements must be stated in GIRAFFE's formal requirement representation language.

### Initial States

Conventional AI planning programs use a fully-specified initial state. All relevant initial conditions are listed in the representation of the initial state. Rather than listing all initial conditions in the initial state of the planning problem, GIRAFFE uses a special kind of operator called *describe* operators to help it find plausible initial states.

Unlike other operators in MOPIE's operator sets, *describe* operators do not represent actions. Instead they describe a set of conditions that commonly occur in a domain. They are called "describe" operators because the conditions they produce have a

common object. For instance, `describe system` produces conditions that represent a certain type of system (see Figure 40). The conditions might be that the system can do I/O at certain terminals, that the system has access to the registrar's database, and so on.

To MOPIE, `describe` operators are no different than other operators used in planning. When MOPIE is looking for an operator to produce a particular condition, it considers the `describe` operators in its operator set in the same way that it considers other operators that represent capabilities of the artifact or its environment.

For example, suppose MOPIE has a problem where a goal or subgoal is a condition "Abigail's schedule is in the registration program's database." Such a subgoal might arise if MOPIE were solving a planning program like the one shown in Figure 39. The condition could be in the initial state, or produced by an earlier action in the plan. If not, then MOPIE adds an operator to the plan that establishes the condition. The operator could be `{enter student's schedule}` or it could be a `describe` operator that describes students and produces conditions such as "The student's schedule is in the registration program's database."

Since many scenarios include actions where a student's schedule is in the database, it is not worthwhile for the planner to find plans that establish that condition every time it is required for a plan. Each time the planner found such a plan it would be spending resources on a problem that had been solved before, instead of solving more relevant problems.

One way to avoid having the planner repetitively solve the same problem is to include those conditions in the initial state of the problem. This is the method that traditional AI planners use to focus planning efforts on the real problem: if the plan for establishing that condition is not interesting, the condition is included in the initial state. Then the planner does not have to add more actions to the plan to make the condition true, and the planner does not do repetitive or irrelevant work to establish the condition.

However, there are several problems with that approach. First, it relies on the client to completely specify the initial state. If the client forgets to include the condition "Abigail's schedule is in the registration program's database" in the requirement, even if it is usually true, then the planner will not find many relevant plans that are made possible when that condition is in the initial state.

A second problem with adding conditions to initial states is that the conditions that are true of some types of objects are not true of others. Students are likely to have schedules in the registration program's database; staff and faculty are not. This kind of distinction is important to GIRAFFE because it considers various object types for objects in requirements and the initial state must reflect the type of an object. The fully-specified initial states used by tradition planning programs do not have the ability to vary initial states in this manner.

One purpose of `describe` operators is to distinguish between objects in a domain. An operator describing a student represents the fact that the student wants to take classes, has transcript information in the registrar's database, and so on, whereas faculty, staff and other subtypes of `person` do not. Because `describe` operators represent this kind of information, they are a good application for knowledge acquisition techniques that acquire classification information. In particular, repertory grids (Boose, 1986) would be a good way to provide automated support for creating the `describe` operators in a domain model and for creating problem-specific `describe` operators. The current version of GIRAFFE uses `describe` operators in its domain model but provides no automated support for creating domain models or for using problem-specific `describe` operators.

Another problem with adding initial conditions is that some conditions might be related. A student might have terminal access, but only if she is at school. Likewise, she might not have phone access at school the way she does at home. GIRAFFE's `describe`



operators provide a way to group related initial conditions together, so that a single condition will not be inappropriately added to an initial state.

A fourth problem with adding conditions to the initial states of requirements is that in some cases they might not be true. Adding a condition to an initial state is a way of making an assumption. In cases where the assumption is not valid, the planner's results will be misleading.

Describe operators produce conditions that are assumptions about an initial state: that the system will have a connection available, for instance. These assumptions are very useful because they let GIRAFFE find interesting scenarios without making it repeatedly find the plans that set up the conditions produced by a describe operator. On the other hand, these assumptions can be dangerous because they will not always be true.

```

name: describe system
description: describe a program
objects: program1 database1 terminal1 loc1

consumed conditions:
program1 is a program (system) that has not been
described

used conditions:
terminal1 is a terminal

produced conditions:
program1 has access to database1
program1 has a connection available
program1 can do I/O at terminal1
terminal1 is not in use
terminal1 is at location loc1

attributes: describe

```

FIGURE 40. A describe operator.

GIRAFFE addresses the dangers of assumptions in two ways. First, it explicitly states them as assumptions when necessary. It does this by adding the conditions to the

initial state of the relevant requirement. Thus the assumptions are brought to the attention of the analyst, who must approve whatever transformations GIRAFFE makes, and thus they are recorded explicitly so that they can be evaluated when the environment of the artifact changes.

The second way that GIRAFFE addresses dangers of assumptions is by deriving plan support and plan obstruction requirements. In analyzing a plan support requirement, GIRAFFE looks at plans that will make a condition hold and can analyze specification changes in terms of that. Similarly, in analyzing plan obstruction requirements, GIRAFFE looks at scenarios that can make assumptions not true. For instance, a network failure scenario might cause program1 to not have access to database1. By deriving and analyzing a plan obstruction requirement for that condition, GIRAFFE can help the analyst determine the usefulness of a given assumption.

### The Planner and Its Results

The plans returned by MOPIE differ from those produced by OPIE in two respects. They include multiple paths, and object types might be more general or more specific than those in the original planning problem.

#### Paths

Unlike OPIE, MOPIE does not stop looking for plans once it has found a plan that achieves the specified final state. Instead it keeps looking as long as it still has viable options for plans. This modification causes two problems because there are an infinite number of plans and it is possible to find duplicate plans.

The problem of infinite numbers of plans is addressed by setting time limits for the planner. Setting the right limits is important because GIRAFFE calls on OPIE to find a large number of plans so the effects of mistakes are multiplied. If the limit is too high then

OPIE might waste time considering useless possibilities. If the limit is too low then OPIE might not find interesting scenarios.

Two heuristics that address the question of limits are:

◆ *Give more time to finding scenarios for safety requirements.* This heuristic is based on observation of MOPIE's performance in GIRAFFE's domain of on-line registration. The heuristic is implemented by assigning a higher default value to time limits of safety requirements than to other kinds of requirements.

◆ *Give more time to finding scenarios for more important requirements.* This heuristic is not automated. The analyst (GIRAFFE's human user) must allocate time according to his or her estimation of the importance of requirements. GIRAFFE provides a means to record the time limit as an attribute of a requirement.

Another mechanism that addresses the problem of infinite plans in MOPIE is pushing constraints into the planner. GIRAFFE uses various constraints to filter out plans that are not interesting for a particular requirement. Constraints that are applicable for all requirements are pushed into the planner so that OPIE will not consider plans that violate those constraints.

The problem of duplicate plans is a difficult one. MOPIE rejects duplicate plans when it discovers them. However, many plans differ only in trivial details but are not detectable as duplicates by MOPIE. For instance, if the order of two actions is reversed in two different plans, MOPIE still considers them to be distinct plans. Considering all possible orders of operators makes the duplicate-checking process more expensive and could also make MOPIE discard plans where the difference in order is significant.

### Object Types

GIRAFFE generalizes object types when converting a requirement to a problem statement. Rather than using the object types specified in the requirement, GIRAFFE replaces each type with the most general type, called *thing*. This allows OPIE to find plans that are more general than it would otherwise find.

For instance, suppose that <transcript privacy> is originally defined as "a student should not be able to find out another student's schedule." GIRAFFE replaces the object type *student* with *thing*, as shown in the left side of Figure 41. During the course of planning, the planner propagates constraints and thereby specializes the object type from *thing* to the most general type possible for a given plan. Some plans for <transcript privacy> might be executable by any *person*, a more general class, and some might be executable by *undergrad*, a more specific class. The right side of Figure 41 shows a case where there is a net generalization.

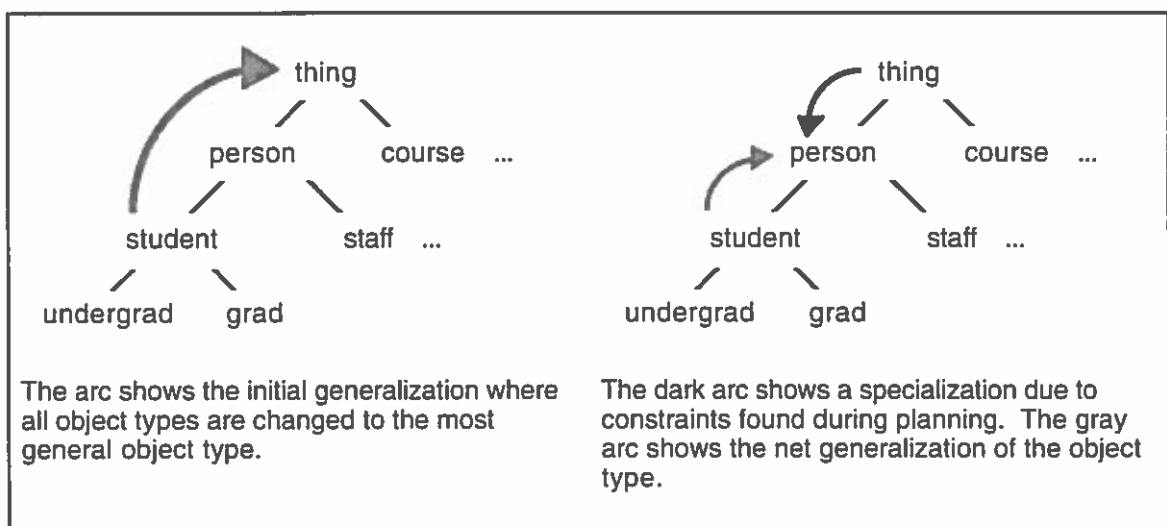


FIGURE 41. Generalization of object types in GIRAFFE.

If the object type in a violation scenario found by the planner is `person` then satisfaction of the requirement is weaker because violations can occur in more situations. If GIRAFFE did not generalize object types before giving the problem to the planner then it would not be able to find the more general violation. If the object type in a violation scenario is `undergrad` then the violation is weaker. MOPIE (and OPIE) would find this plan even if GIRAFFE did not generalize all objects.

### Other Methods for Finding Scenarios

GIRAFFE currently uses planning methods to find scenarios. In this section I discuss other approaches to finding scenarios, including forward chaining methods, acquisition methods and case-based methods. These methods complement planning methods and are more useful than planning for finding certain kinds of scenarios such as failure scenarios and some complex scenarios given by domain experts. Therefore, future versions of GIRAFFE might incorporate techniques similar to the ones discussed here.

Previous sections of this chapter describe GIRAFFE's mechanisms for finding general scenarios. In this section I describe alternative methods. As I discuss each method, I compare it with MOPIE's planning methods.

### Forward Chaining

In this section I discuss methods that use forward-chaining and projection to find scenarios. Rather than considering what actions are necessary to achieve a final state, programs that use forward chaining consider what actions are possible given an initial or intermediary state. Projection is similar to forward chaining in that it considers what conditions hold after execution of an action.

I describe two programs in this section. The first, called SBRE, uses forward-chaining to find scenarios that can occur from a given initial state. The second program

uses projection to determine the possible scenarios that can occur from execution of a given plan.

Kaufman et al.

Kaufman et al describe Scenario-Based Requirements Engineering (SBRE) which uses a forward-chaining approach to finding scenarios. Since SBRE has an initial state given, it has a final-state problem analogous to GIRAFFE's initial-state problem. SBRE must determine what final states are interesting. It does this by giving control to the analyst (the program's user)—when there is more than one rule that can fire, the analyst chooses one. The analyst thus develops one scenario at a time. In contrast to the single-path process used in SBRE, GIRAFFE considers multiple paths.

In addition to choosing between several rules that can fire, the analyst can enter values for variables. Entering values in that way allows the analyst to investigate "what-if" cases. For example, in the on-line registration domain, the analyst might consider the case where a network link is down. She would enter the appropriate value for a variable and then see what actions occurred. Instead of having an analyst enter values, GIRAFFE finds such scenarios by using environmental actions to produce conditions such as network down.

Although the SBRE approach does not include generalization mechanisms, Kaufman (1988) discusses the use of scenario coverage techniques which play a similar role in some respects. GIRAFFE attempts to find scenarios that cover the most general objects possible, while SBRE attempts to show that all relevant object types (for example) are covered by scenarios. SBRE increases the number of scenarios where GIRAFFE tries to increase the scope of a general scenario.

SBRE's scenario-generation program is useful for analyzing "what-if" cases but is less effective for analyzing "how-could" cases. If the analyst wants to see whether a

certain state can occur, she must enter values and select rules that she thinks will lead to that state. GIRAFFE can use the planner to find many "how-could" cases, but others are more difficult. For example, if a client wants to know whether it is possible for a student to correctly execute a series of actions and still not be registered for a class, then neither GIRAFFE's planning methods nor SBRE's forward-chaining are suitable. One way to find such scenarios is projection.

### Hanks

Hanks (1990) describes a program that projects plans with actions that have probabilistic outcomes. Each action's effects are described in terms of one or more outcomes with a probability assigned to each outcome. The projector produces scenario trees with branches to represent undetermined outcomes. Figure 42 shows a simple example of a scenario tree. There are two branches, where the results of the action are undetermined. When the student calls the registration system, she might get a busy signal and might get a phone connection. Since it is not possible at planning time to determine which condition will occur, there is a branch in the tree.

Each path through the scenario tree is called a chronicle. Since each action can have several outcomes, producing a full scenario tree is impractical. Instead, the program bundles chronicles together so that all chronicles within a single bundle have no important distinctions.

Projection is useful for analyzing problems where the client would like to know how a plan can fail. The plan failure problem is a difficult one to analyze with a planner alone because there are an infinite number of ways to *not* achieve a desired state. Of that infinite number, only a few plans will be interesting. By using a projector in addition to a planner, a program can find plans that are "close" to a successful plan but still fail. Close

can be defined in terms of how many actions are common to the successful plan and the failed plan.

Instead of using a projector that analyzes actions with more than one possible outcome, GIRAFFE could use a projector that analyzes whether or not a particular action occurs. Such a projector would actually be more like a plan breaker because it would try to add or delete actions that cause the plan to fail without taking the plan outside given bounds. For instance, a client might ask if there's any way that adding a class could fail if a student follows the correct procedure: calling the registration program, logging in, and entering the class information.

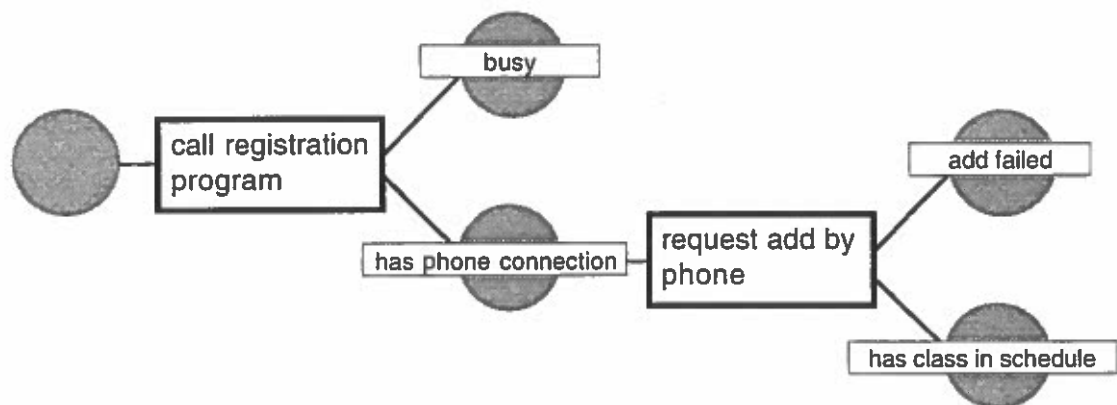


FIGURE 42. A plan fragment showing possible outcomes of actions.

The plan breaker would not remove any of the actions given in the plan. Instead it would add actions or change conditions to try to make the plan fail. For instance, it might add an action where another student takes the last seat in the class, thus causing the plan to fail. Or, it might add an action where the system fails. Thus it would create a new failure scenario from an existing scenario. This type of plan breaker is not part of GIRAFFE but could later augment its scenario-finding capability.



## Acquisition

An alternative to using planning methods or forward-chaining techniques, is for a program to acquire scenarios from a human. In this section I describe two projects that use acquisition techniques in requirements engineering. The first program I describe is called TAMS. It addresses problems in merging scenarios. The second program, called ISAT, uses a form of explanation-based generalization to generalize acquired scenarios.

### Dardenne

Dardenne (1993) proposes a system called TAMS (Tool for Acquiring and Merging Scenarios) for acquiring scenarios. TAMS is a domain-specific tool used by an analyst to record and analyze scenarios described by a client. TAMS is based on the idea that people naturally use scenarios to state requirements and the idea that those scenarios must be merged or made consistent.

TAMS' representation for scenarios includes actions in the scenario with combination modes (sequential, parallel, alternative, repetitive and undefined). In addition to actions, scenarios in TAMS also have an initial state made up of predicates that must be true for the scenario to execute. Scenario elements, such as actions and predicates, and scenarios themselves are represented in the meta-model of KAOS (Dardenne, van Lamsweerde, & Fickas, 1993).

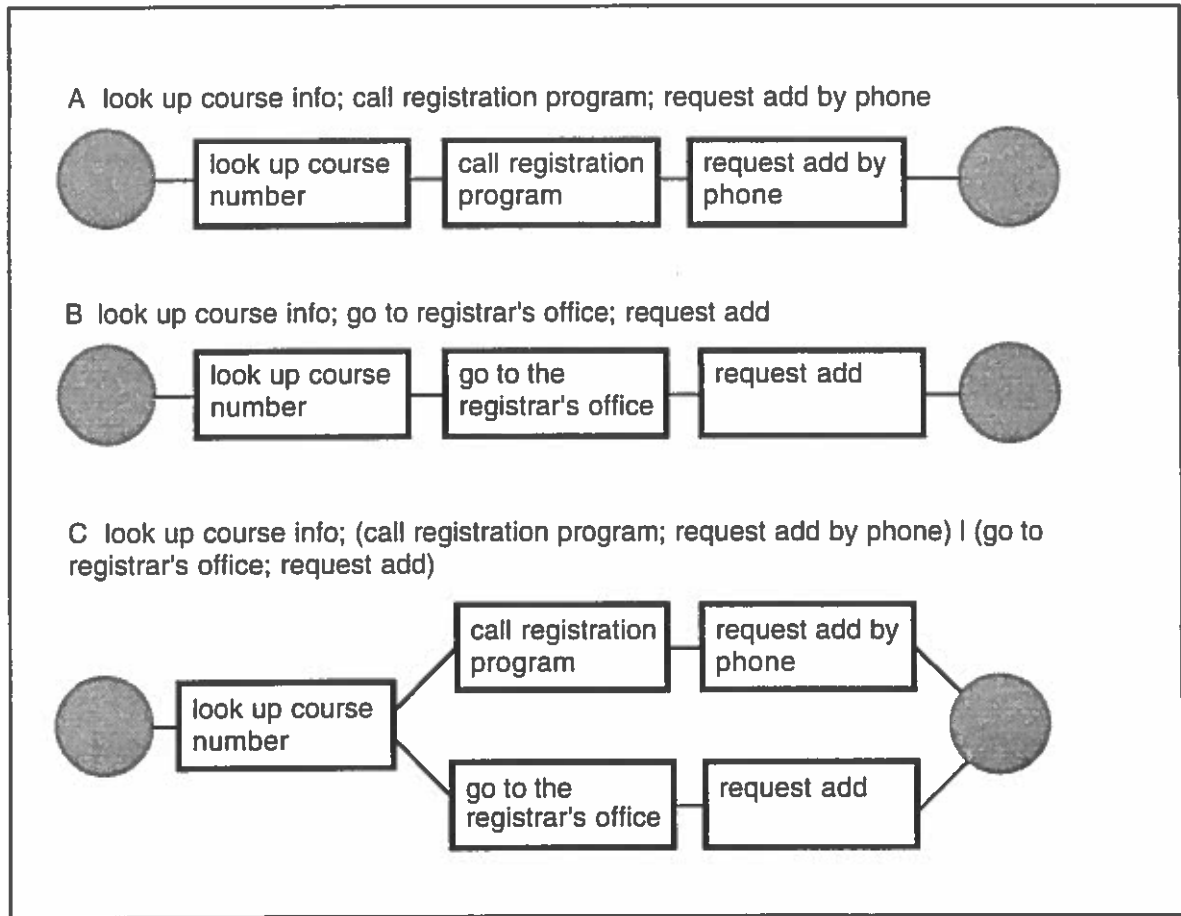


FIGURE 43. Merging scenarios in TAMS. Scenarios A and B are merged to produce C.

In merging scenarios, TAMS resolves conflicts between actions and combination modes. So, for instance, if one scenario described a student registering a class using phone registration and another scenario described a student registering in person, TAMS could merge the two scenarios by combining the phone `request add` and `request add` actions (or a longer sequence of actions) with the alternative combination mode, as shown in Figure 43. GIRAFFE would handle a similar situation by using an abstract action to represent the choice of actions.

TAMS will also address issues in merging positive and negative scenarios (or achievement and safety transitions, in GIRAFFE's terminology). GIRAFFE uses global

rating functions to find inconsistencies between the two kinds of scenarios but does not create a single scenario that merges the two.

No facility specifically for generalizing scenarios is described for TAMS, although in some cases a merged scenario can be considered more general than the scenarios from which it was created.

### Hall

Hall's ISAT system (Hall, 1993) acquires scenarios from its user and generalizes them. ISAT represents scenarios as a sequence of events and observations. It uses simulation to determine satisfaction of requirements (whether scenarios successfully execute or not) and so is well-suited for analyzing failure requirements, whereas GIRAFFE is limited in that respect. For example, if ISAT worked in GIRAFFE's domain, a user might enter a scenario where a student calls the registration program, logs in and requests an add. An observation at the end of the scenario would check to see if the class is actually in the student's schedule.

ISAT uses a form of explanation-based generalization to generalize the scenario. It does not use hierarchies of objects and actions as GIRAFFE and MOPIE do, so it cannot generalize object types as GIRAFFE does. Instead it replaces objects with variables in a way that is guaranteed to be sound. In this context, soundness means that the generalized scenario will succeed (i.e., the expected observations will occur) in the same situations that the concrete one will.

ISAT has no abstract action types and so cannot describe scenarios in terms of more general action types as GIRAFFE can. Furthermore, since the only constraints on scenarios are observations of state, there is no way to state any path constraints on actions.

ISAT can suggest additional "gap-filling" scenarios and so in some sense reasons about multiple paths as GIRAFFE does. However, gap-filling scenarios are derived from a

scenario's initial state and will not be arbitrarily different from the original scenario. By contrast, MOPIE can find multiple paths with initial states and actions that are completely different.

### Case-Based Planning

Another approach to finding scenarios is to use case-based planning. This approach combines acquisition (or some other means of obtaining an initial set of plans) with retrieval and adaptation mechanisms.

Adaptation techniques include refinement and repair. MOLGEN () was an early case-based planning program that refined skeletal plans to find executable concrete plans. Another technique is to adapt concrete plans to a new situation. In this section I describe CHEF, which uses repair techniques, and SCARE, which uses refinement techniques.

#### Hammond

Hammond (1989) describes a program called CHEF that uses case-based planning. CHEF has a set of concrete plans with general indices. CHEF uses its indices to retrieve the plan that it considers most relevant and then adapts the plan using its object critics and repair strategies.

Suppose that CHEF worked in GIRAFFE's domain of on-line registration. In its initial set of plans it might have a plan called UG-ADD for an undergraduate to add a class to her schedule. Then suppose that CHEF is asked to find a plan for an undergraduate to register for a class that requires the instructor's consent. Using its indices, CHEF retrieves a plan from its case base, such as UG-ADD. Since UG-ADD has no step for the student to get the instructor's consent, UG-ADD will fail. CHEF's object critics catch this kind of problem and suggest modifications.

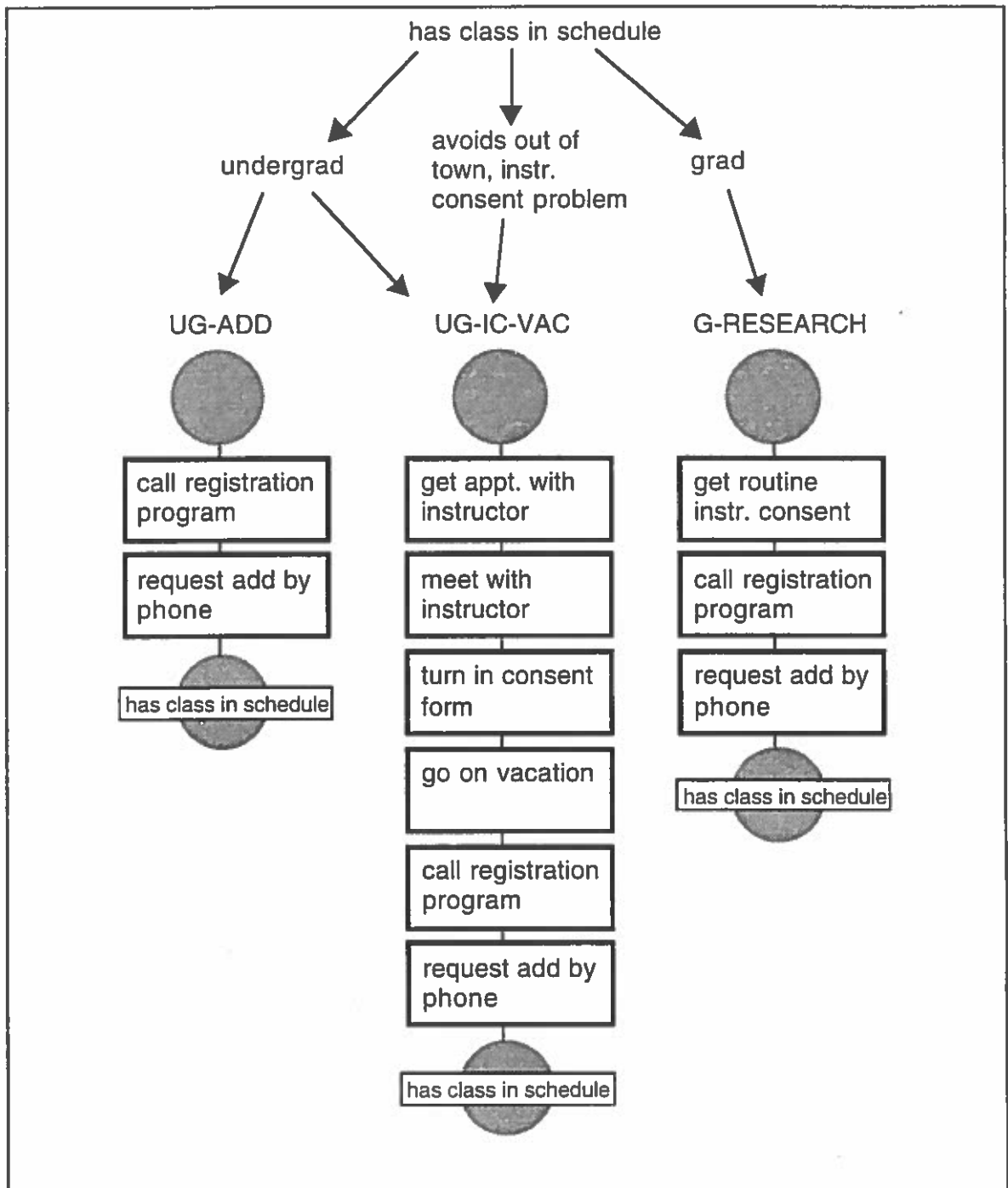


FIGURE 44. Indices to plans, a la CHEF.

Object critics find problems due to changes in objects in the plan, such as the change in course types in the example. They do not address problems with goal interaction, which is the purpose of CHEF's TOPs. For instance, if a student has plans to travel in Europe and also to register for a class she could use phone registration with relatively little goal interaction (unless a goal is a low phone bill) but if the class requires instructor consent the plan requires additional modification and the student might need to make special arrangements with the instructor before leaving on her trip.

Once CHEF has solved a problem using its repair strategies, it keeps a record of the problem and solution as a way of anticipating and avoiding future problems of the same nature. The revised UG-ADD plan, which I'll call UG-IC-ADD, is added to CHEF's knowledge base of plans.

Because of its bias toward anticipating problems, CHEF sometimes passes up plans that seem more relevant for plans that are less similar in some respects but have the same interaction problem. Suppose CHEF had a plan for a graduate to add a research course, which requires the instructor's consent, called G-RESEARCH in its case base, as shown in Figure 44. If asked to find a plan, called G-VAC, for a graduate to go on vacation and add a course, CHEF might pass up G-RESEARCH in favor of UG-IC-ADD because the latter has the same goal interaction of going on vacation and registering for an instructor's-consent class.

In some cases the method of retrieving plans because of similar failure indications seems to lead CHEF to unsuitable choices. In the G-VAC plan, the student might routinely get permission for a class whereas in the UC-IC-ADD plan the student might have to get an appointment with the instructor and make other arrangements that aren't necessary in G-VAC.

Although some of CHEF's capabilities would be useful for finding scenarios for GIRAFFE, its general approach of storing concrete plans and using general indices would

not be appropriate because of GIRAFFE's use of general scenarios. An approach that stores and retrieves general plans is more suited to GIRAFFE because of GIRAFFE's use of general scenarios. CHEF requires concrete plans as solutions to problems, so it would have to specialize in every case if it stored general plans. GIRAFFE, however, uses general plans and so would have to generalize in every case if it stored specific plans.

A further limitation of CHEF's techniques is that they were developed for single-agent domains such as CHEF's cooking domain. CHEF's problem classifications and repair strategies don't account for the interactions between agents that are important in other domains. Two systems that address such issues are described in the next section.

#### Fickas et al.

Skate (Fickas & Nagarajan, 1988) is another program that uses a case base to find scenarios. Skate's scenarios include interactions between agents, such as when one person intimidates or otherwise influences another. Skate's case base in the library domain includes a scenario where one library patron intimidates another in order to get access to a book that's checked out. An example in the registration domain could be called "priority swapping".

In a priority-swapping scenario, a student, Abigail registers for a class that another student, Basil, wants to take. Abigail has higher registration priority and is thus more likely to get the class. She arranges with Basil to drop the class just before Basil adds it, leaving a seat available so that Basil's add is successful. In effect, Abigail gives Basil the benefit of her higher priority.

Skate has some ability to adapt scenarios but has no way to store new scenarios as CHEF does. Furthermore, it faces a difficult problem in matching scenarios with specifications. Helm and Fickas (1992) propose a system call SCARE that uses failure scenarios to analyze and critique specifications. It overcomes Skate's matching problem by

associating abstract failure scenarios with transformations that are used to derive specifications. When a transformation is applied to the specification, SCARE adds the abstract scenario associated with the transformation to the relevant scenarios for the specification. Then it refines the abstract scenario to a scenario that fits the specification using plan refinement techniques such as those used by MOLGEN and OPIE.

Some scenarios that are difficult for MOPIE to find because of their length or complexity could be attached to actions in the domain model. For instance, some registration systems include a department enrollment override that allows a particular student to enroll even though a class already has a maximum number of students enrolled. Suppose that Abigail and Basil agree to exchange sections of a class in a scenario that is similar to the priority-swapping scenario except that both students begin with the same class in their schedule. If an enrollment override is in effect then when Abigail drops the class there are no seats available and Basil's add will fail.

Using SCARE's technique of associating failure scenarios with specification elements, GIRAFFE could associate a general failed-exchange scenario with the enrollment override action. When the enrollment override action is added to the specification is added to the specification, GIRAFFE's planner would attempt to refine the general plan and if successful would find a relevant scenario that would be difficult to find with GIRAFFE's current methods.

### Summary

GIRAFFE uses planning techniques to find the scenarios it uses to transform requirements. The planner that finds scenarios is MOPIE, a modified version of the OPIE planner. MOPIE finds plans that are more general than those found by OPIE in that they include multiple paths and more general (or more specific) object types than those in the



original planning problem. MOPIE also uses `describe` operators to find relevant conditions for the initial states of requirements.

Although GIRAFFE currently uses only planning techniques, future versions could use other methods of finding scenarios. In this chapter I discussed several methods that other programs use, including forward chaining, acquisition and case-based planning.

## CHAPTER VI

### EVALUATION

#### Introduction

In this chapter I evaluate the method of requirements transformation defined earlier in this dissertation. In the first part of the chapter I evaluate the implementation of the method by showing its ability to rationalize requirements of three existing artifacts. I describe the three artifacts and GIRAFFE's analysis of them and give two experts' comments on GIRAFFE's analysis.

In the second part of the chapter I evaluate the generality of the implementation. I discuss the domain-dependence of GIRAFFE's knowledge base and describe the characteristics of domains where GIRAFFE can effectively transform requirements.

Evaluation of the implementation provides some evaluation of the method as a set of general principles and heuristics. In the third part of this chapter I discuss the degree to which the program's strengths and weaknesses reflect on the general method.

#### Evaluation of the Implementation

In this section I evaluate the implementation of my requirements transformation method in the GIRAFFE program. I begin by stating the criteria used to evaluate GIRAFFE. Then I describe how I consulted two experts to help evaluate the program.

After describing the evaluation criteria and the domain experts, I give the initial requirements used by GIRAFFE in analyzing three artifacts, referred to here as Artifacts A-C. Artifact A is a system with no support for phone or terminal registration, Artifact B

includes phone registration only and Artifact C includes terminal registration only. I evaluate GIRAFFE's transformation of the initial requirements for each artifact.

All three artifacts are (or were) implemented artifacts. However, GIRAFFE's analysis of them is based on my interpretation of the systems' documentation and does not necessarily apply to the actual system. For each artifact I summarize GIRAFFE's analysis and the transformations that it suggests. Then I discuss the strengths and weaknesses of the program's analysis.

### Evaluation Criteria

The purpose of the GIRAFFE program is to support intertwining by finding the strongest requirements satisfied by a specification. I evaluate GIRAFFE according to how well it accomplishes that purpose. GIRAFFE succeeds in cases where it weakens a requirement because the requirement was not satisfied by the specification and in cases where it strengthens a requirement that is still satisfied after the transformation.

GIRAFFE does not succeed in the following cases:

- ◆ no representation

When GIRAFFE has no adequate representation for a requirement it cannot analyze it and therefore fails to achieve its purpose. To analyze GIRAFFE in this respect I discuss its coverage of requirements issues and asked the domain experts to do the same.

- ◆ too weak

If there is a stronger requirement than the one GIRAFFE finds then GIRAFFE has not found the strongest satisfied requirement.

- ◆ too strong

If the requirement that GIRAFFE finds is not satisfied by the specification then GIRAFFE has failed with respect to that requirement.

- ◆ inconclusive or misleading ratings

In some cases GIRAFFE must rely on human judgment or preference to select transformations since its abilities to rating requirement strength are limited. In such cases GIRAFFE fails to fully accomplish its purpose but it succeeds to the extent that it can separate viable alternatives from superfluous ones.

Next I describe the domain experts and then I evaluate GIRAFFE's capabilities in terms of the criteria stated in this section.

### The Experts

To help in the evaluation of GIRAFFE, I interviewed two people with experience with on-line registration systems. I will refer to one of the two people as the domain expert and to the other person as the analyst. The domain expert gave comments from the point of view of a client. She has participated in defining requirements for on-line registration systems and general experience in a registrar's office. The other expert gave comments from the point of view of an analyst and software developer. Most of the analyst's experience was in other domains but he has spent a year working in the domain of on-line registration .

I gave each expert a set of transformations from GIRAFFE's analysis, along with a description of the terminology and notation used in the examples. I also gave them descriptions of the specifications for each artifact (A-C) and statements of the complete requirements set before and after GIRAFFE's transformations. The notation for the examples and requirements sets was a text version of GIRAFFE's formal representation. Although informal, the notation referred to the types of objects used in GIRAFFE's formal representation: initial conditions, final conditions, actions, and so on.

I asked the experts general questions about their work in requirements engineering and asked them to answer the following questions as they looked at each example:

1) Is there a requirement, in an example or in a requirements set, that is not satisfied by the corresponding specification? In other words, is there a requirement that is too strong?

2) Is there a requirement, in an example or in a requirements set, that is not the strongest requirement satisfied by the corresponding specification? In other words, is there a requirement that is too weak?

I also asked the experts to give other comments, including comments regarding:

- the format of the requirements or the specification summaries
- issues that should have been addressed but were not
- issues that are superfluous or given more importance than they deserve

I discuss specific responses by the experts in later sections that describe each artifact. The expert's general comments were as follows:

When asked whether requirements change during the course of software development, both experts described situations where requirements change. The domain expert told about stages of development where an initial system was enhanced and requirements changed for each new stage. She described the requirements engineering process as one of constant change and refinement.

The analyst said that requirements become weaker when a developer discovers that something is too costly or otherwise impractical to implement. On the other hand, he noted that they become stronger when the client realizes that additional features are possible.

Both the domain expert and the analyst expressed uncertainty over the concept of stronger and weaker requirements as used in the descriptions of GIRAFFE's example transformations. Although they later understood how those terms are used, it was easier for them to give comments on the changes in requirements in general terms. Therefore in

the following sections I give their remarks on various requirements and changes in general terms rather than in terms of stronger and weaker requirements.

In the next section I describe the initial requirements that GIRAFFE used in its analysis of all three artifacts, and give comments by the experts on the initial requirements and missing requirements. Then I describe GIRAFFE's transformation of those initial requirements for each of the three artifacts.

### Initial Requirements

GIRAFFE uses a standard set of requirements to begin its analysis. The initial requirements are part of the domain-dependent information that GIRAFFE requires. For each of the three artifacts discussed in this chapter GIRAFFE used the following initial requirements:

- <add class> achievement

A student should be able to add a class to her schedule for a term.

- <find out schedule> achievement

A student should be able to find out what classes are in her schedule for a given term.

- <unwanted class> safety

A student should not have an unwanted class in her schedule.

- <get advice> achievement

A student should be able to get advice on what classes to take in a given term for a given degree.

- <get transcript> achievement

A student should be able to get an official copy of her transcript.

- <get transcript info> achievement

A student should be able to find out her academic history.

- <get school info> achievement

A student should be able to get information typically found in the school bulletin and class schedule, such as registration dates and places, class times and places, degree requirements, and so on. This requirement represents all such information as a single abstract information item because distinguishing the various types would add complexity to the model without demonstrating significant strengths or limitations of GIRAFFE.

- <get help info> achievement

A student should be able to get help with the registration process.

- <inaccurate student info> safety

A student should not have inaccurate information in the school's database, such as name, phone number, address or other information.

These requirements, as well as GIRAFFE's rules for deriving and transforming requirements, are based on consultation with domain experts and papers in the proceedings of a series of conferences on enhancing academic support services (Kramer & Petersen, 1991). GIRAFFE can analyze other requirements that are part of the domain provided that they are stated in its representation language.

The requirements in the initial set are simple in that their transitions have no qualifications. There are few initial conditions or path constraints, and those that are present are necessary to define the transition.

In discussing GIRAFFE's requirements and transformations, the domain expert stated three safety requirements that were missing from GIRAFFE's initial set. Those are:

- <maximum enrollment> safety

No student can be registered for two sections of the same course.

- <no over-enrollment> safety

No course should exceed its enrollment limit. In Artifact B this requirement is weakened so that some kinds of classes can be over-enrolled if a department overrides the limit.

- <last course> safety

A student cannot drop her last class without going to the registrar's office. Note that this requirement requires a path constraint: "without going to the registrar's office". The path constraint is not a qualification that can be removed to make the requirement stronger. Instead, the path constraint defines a transition of interest and so is a definitive path constraint.

Although the three additional safety requirements listed above were not in GIRAFFE's original set of requirements, they can be represented and analyzed using GIRAFFE's knowledge base. A type of requirement that does not fit well into the representation is a requirement for Artifact B that a response should be given within two seconds after a student enters a transaction code over the phone. GIRAFFE can compare approximate durations but its representation is not well-suited to the kinds of calculations required to analyze the two second requirement. Its domain model does not have values sufficiently accurate to determine whether or not such a performance requirement would be satisfied.



### Artifact A—No Telephone or Terminal Registration

In this section I summarize GIRAFFE's analysis of the requirements for Artifact A and discuss the strengths and limitations of that analysis. The specification for Artifact A does not include capabilities for touch-tone telephone or terminal registration. Students must be at a specific place to register for classes and make other transactions.

#### Summary of GIRAFFE's Transformations

In this section I list each of the new requirements that GIRAFFE derived for Artifact A, except for the failure requirements. Transformations that derive failure requirements and transformations that weaken or strengthen requirements are too numerous to list individually so I summarize them after the list of derived requirements.

GIRAFFE derives the following requirements for Artifact A that were not part of the initial requirements:

#### PRIVACY

##### ■ <transcript privacy>

No one else should find out the information in a student's transcript. This requirement was derived from <get transcript info>.

##### ■ <advice privacy>

No one else should find out advice given to a student. This requirement was derived from <get advice>.

##### ■ <schedule privacy>

No one else should find out a student's schedule. This requirement was derived from <find out schedule>.

## PLAN SUPPORT

### ■ <get id>

A student should be able to get an ID card. This requirement was derived from <get schedule info>.

### ■ <seat available in class>

Classes should have space available in them. This requirement was derived from <add class>.

## REPAIR

### ■ <drop class>

A student should be able to remove a class from her schedule if she doesn't want it. This requirement was derived from <unwanted class>.

### ■ <change student info>

Someone should be able to update a student's information in the database. This requirement was derived from <inaccurate student info>.

GIRAFFE weakens the achievement requirements to show that students must go to a certain place to use most of the functionality of the artifact. For registration transactions they must be at the registrar's office and to get advice they must be at an advisor's office.

GIRAFFE also weakens the achievement requirements to show that there must be someone working at the registrar's office (i.e., the office must be open) and the proper information must be in the registrar's database.

GIRAFFE weakens the privacy requirements by stating that information will not be private if students tell the information or give documents with the information to another person. GIRAFFE also indicates that information will not be private if students give their ID to someone else, lose their ID, or if someone steals their ID.

### Strengths and Weaknesses of GIRAFFE's Analysis

Although the initial versions of the achievement requirements are similar, GIRAFFE uses the specification to transform each requirement in different ways. For example, GIRAFFE states a stronger requirement for <get schedule> than it does for <get transcript info> since the specification indicates that students don't have to request a copy of their schedule but they do have to request a transcript.

In analyzing privacy requirements, GIRAFFE states what constraints must be true of the environment to prevent violations. It prefers more specific constraints, such as "a student doesn't tell her transcript information" to "a person doesn't tell someone's transcript information." It also prefers constraints on people whose information would be disclosed over constraints on arbitrary agents in the environment.

An example that shows both the strength and weakness of GIRAFFE's rating functions is its analysis of path constraints on the <transcript privacy> requirement. Scenarios that show violations of this requirement include various actions by agents in the environment and GIRAFFE considers constraints on each of them.

The strength of GIRAFFE's rating functions show in the rating they assign to a constraint on the {give an id} action, which receives the highest rating of the candidate transformations. The high rating is due to the fact that in the violation scenarios {give an id} is performed by the student whose information should be kept private. GIRAFFE considers such an agent to be motivated to keep information private and therefore gives that constraint a higher rating than the constraint on {steal an id}, an action performed by an arbitrary agent with no motivation for preserving the privacy of the information.

GIRAFFE distinguishes between {give an id} and {give a transcript}. The former is more important because giving an ID can lead to violations of other privacy requirements

so GIRAFFE rates the constraint on {give an id} higher than the constraint on {give transcript}, which is important only for <transcript privacy>.

The limitation of GIRAFFE's rating functions show in the <transcript privacy> example because GIRAFFE assigns the same rating value to {go to the registrar's office} that it does to {give an id}. The constraint on {go to the registrar's office} is not justified by comparison with implemented artifacts because few of them contain restrictions on who goes to the registrar's office. GIRAFFE decreases the rating on {go to the registrar's office} because it appears in achievement requirement constraints but increases it because it appears in violation scenarios of other safety requirements, such as <schedule privacy>.

GIRAFFE is able to rule out some related requirements that are unlikely to be of interest. For instance, it doesn't suggest privacy requirements for every achievement requirement because it only looks for those where someone knows information in the final state. It distinguishes between someone finding out public information, such as school information, and personal information, such as transcript information. Its ability to rule out spurious privacy requirements is one of its strengths.

GIRAFFE's ability to derive repair requirements is somewhat more limited. For instance, in the case of <inaccurate student info> there is no way of satisfying the requirement short of putting unreasonable constraints on the environment. Because the requirement is not satisfied, GIRAFFE derives a repair requirement, which is an appropriate action. However, GIRAFFE suggests two requirements: one where the information is updated in the school database and one where the student's information changes. The second is a spurious requirement because it amounts to asking a student to move back to her old address (for example) and is not reasonable. Currently GIRAFFE suggests both requirements and relies on human judgment to distinguish between them.

The usefulness of some requirements derived by GIRAFFE is difficult to determine. One such case is when GIRAFFE suggests a plan-support requirement to

achieve the condition `seat_available`. At first such a requirement might seem unreasonable since the most obvious way to achieve it is for another student to drop a class which would help one student at the expense of hurting another.

One way to partially satisfy the seat-available requirement is the "administrative drop," where students who don't attend a course that is in high demand are automatically dropped. Another alternative is to derive the related requirement but then weaken it so that instead of achieving a condition where a student is available it achieves a condition where students know when a seat becomes available. Such a weakening rationalizes capabilities which help students find open sections of a class. That transformation would also rationalize an artifact that notified students when a course was available. Although Artifact A does not have this capability, a notification capability would be feasible in an environment where electronic mail is accessible to all students.

The second alternative for satisfying the seat-available requirement is beyond the scope of GIRAFFE's current domain model and requirements transformations. GIRAFFE suggests the requirement but in this case relies on human judgment to determine whether it is relevant. Thus GIRAFFE's ability to assess the usefulness of related requirements varies from privacy requirements where it is unlikely to suggest superfluous requirements to suggesting repair requirements where it does suggest superfluous requirements in at least one instance.

In discussing transformations that strengthen and weaken requirements, both experts suggested additional qualifications for `<add class>`. The domain expert suggested eligibility to register, registration priority, and several more obscure qualifications. The analyst mentioned courses that require instructor's consent and fees. The problem of determining which conditions should be represented as qualifications to the requirements is an instance of the general qualification problem (Ginsberg & Smith, 1988) in AI. Although

it is always possible to state additional conditions, there is a useful set that can be represented in GIRAFFE's domain model.

The domain expert noted the importance of <schedule privacy> and the other privacy requirements. Those requirements become even more important when telephone registration is available. In the next section I look at transformations of requirements for an artifact that includes telephone registration.

### Artifact B—Touch-tone Telephone Registration

In this section I discuss GIRAFFE's analysis of the requirements for Artifact B. The specification for Artifact B includes capabilities for touch-tone telephone registration. The specification also includes many of the capabilities of Artifact A, so that students have the choice of registering by phone or in person.

#### Summary of GIRAFFE's Transformations

GIRAFFE derives the same types of requirements for Artifact B that it does for Artifact A. With the inclusion of telephone registration capabilities in the artifact specification GIRAFFE also derives the following requirements:

##### PRIVACY

- <password privacy>

No one else finds out a student's password. GIRAFFE derives this as a privacy requirement but it can also be considered a plan-obstruction requirement because disables a precondition for plans that violate safety requirements.

##### PLAN SUPPORT

- <knows password>

A student can find out her password.

- <has phone access>

A student can get phone access.

#### REPAIR

- <repair password privacy>

If someone finds out someone else's password then there should be a way to make a password private again (by assigning a new one).

In addition to deriving new requirements, GIRAFFE transforms them. In transforming <password privacy> it suggests constraints on actions where office personnel tell passwords without checking ID.

#### Strengths and Weaknesses of GIRAFFE's Analysis

The requirements that GIRAFFE derives are reflected in the implementation of Artifact B. The importance of the <has phone access> requirement is shown in Artifact B by the fact that phones for accessing the registration system were placed by the registrar's office.

GIRAFFE bases derivation of repair requirements on whether or not there is an action that produces the repair condition. It suggests <repair password privacy> but does not suggest a "repair transcript privacy" requirement. In the case of passwords there is such an action, namely {change password} and in the case of transcripts there is no such action. Thus GIRAFFE makes the right decision in deriving the password privacy repair requirement and not deriving the transcript privacy repair requirement.

The domain expert noted that password privacy becomes even more important when additional information is available by phone. She told how one artifact was developed in stages where transcript information was not initially available. When the new capability for giving grades by phone was added, password privacy became even more

important. GIRAFFE's rating functions represent this kind of information by looking at the number of safety violations caused by a particular action, such as {tell password}. The program rates constraints on actions that cause violation of more safety requirements higher than it rates constraints that cause fewer violations.

The domain expert also noted the importance of the plan-support requirement <knows password>. She gave an example where a student left for Europe and called on the way but had forgotten his password and so could not register. The registration system that the domain expert works with, which is similar to Artifact B, allows students to change their password to something that is easy to remember as a way of preventing that kind of problem.

The same registration system assigns passwords that are difficult for intruders to guess, as a means of promoting password privacy. However, the domain expert said that unauthorized access is uncommon. What is more common is for students to deliberately give their password to someone else. Violations of password privacy that occur when a student tells her password to someone else are preventable by the student, which GIRAFFE recognizes in its analysis of the requirement.

The domain expert said that phone registration has limitations, such as not having the capability for students to enter an address or read information, but is still preferable to terminal-based registration because it allows near-universal access. GIRAFFE shows the limitations of phone registration in its analysis because it is unable to derive stronger requirements for <inaccurate student info> (i.e. changing addresses). It shows the strength of phone registration by deriving stronger requirements for <add class>, etc. In the next section I discuss how the requirements that GIRAFFE derives for terminal-based registration differ from those derived for phone-based registration.



### Artifact C—Terminal Registration

Artifact C is derived from Artifact A by adding capabilities to the specification for students to register and get information by computer terminal. The specification for Artifact C does not include phone-registration capabilities.

#### Summary of GIRAFFE's Transformations

GIRAFFE derived the same password-related requirements for Artifact C that it did for Artifact B. It also derives the following requirement that is similar to Artifact B's <has phone access> requirement:

##### PLAN SUPPORT

- <has terminal access>

GIRAFFE strengthens the <change student info> requirement because addresses, phone numbers, and so on can be entered at a terminal whereas it is virtually impossible to enter such information using touch-tone telephone.

For the <get advice> requirement, GIRAFFE introduces a disjunction in the initial state when the client adds an action that allows students to enter their transcripts to the advice program. A student can type in the necessary transcript information if she knows it or the program can get the information from the database if it has access to the database and the information is there.

#### Strengths and Weaknesses of GIRAFFE's Analysis

GIRAFFE's ability to introduce disjunctions in the initial state let it state stronger requirements than it would otherwise be able to do. Since Artifact C has both terminal-

based and non-terminal-based capabilities, many of its requirements have disjunctions in the initial state that represent the choice of using or not using a terminal to register.

GIRAFFE is able to analyze the implications of object generality. In the implementation of Artifact C, the registration system supported any terminal supported by the campus network. The actions in Artifact C's specification accordingly work for any terminal type. However, GIRAFFE weakened other requirements because of this characteristic of the specification.

In discussing Artifact C, the analyst noted that <get school info> is stronger for terminals. The domain expert gave an example of a system that gives schedule and bulletin information (called "school information" in GIRAFFE's representation) by terminal but not personal information such as schedules or grades. Both the analyst and the domain expert noted the importance of maintaining the information.

GIRAFFE's current domain model allows it to strengthen <get school info> more for a terminal system than for a phone system, but does not represent the maintenance requirement. However, that requirement would be similar to the <inaccurate student info> and <change student info> requirements that GIRAFFE's model does currently include.

### Summary

According to the evaluation criteria stated at the beginning of this section, GIRAFFE does not succeed when it cannot represent a requirement, when it transforms a requirement so that it is too weak or too strong, or when its rating functions give misleading or inconclusive results. In this section I showed how GIRAFFE transformed requirements for three specifications and evaluated GIRAFFE's analysis of the requirements.

GIRAFFE can represent and analyze many kinds of requirements. However, it is not well-suited for analyzing performance requirements, other than in qualitative or

approximate terms. In particular, the two-second response requirement of Artifact B is a difficult one for it to analyze. GIRAFFE's ability to analyze satisfaction of failure requirements is limited, although it can suggest and represent them.

GIRAFFE's domain model is incomplete. Although any model is an abstraction and thus not entirely complete, GIRAFFE's model lacks some elements that are within the scope of its representation. Thus its analysis of requirements for the on-line registration is somewhat simpler than it should be. GIRAFFE's model does allow it to represent and reason about a significant number of important issues in on-line registration.

In some cases GIRAFFE derived requirements that were too strong. An example of that is where GIRAFFE suggested constraints on the action {go to registrar's office} which was not in fact constrained in the artifacts. In general, when GIRAFFE failed, it did so by too heavily constraining actions for safety requirements or by missing qualifications for achievement requirements. The latter is an issue primarily for representation of the domain model.

GIRAFFE derived privacy requirements, support/obstruction requirements, repair requirements and failure requirements. In a few cases GIRAFFE derived spurious requirements, such as the repair requirement that required students to undo their address changes rather than updating the database, but most of its derived requirements were useful.

GIRAFFE succeeded by weakening the initial requirements to produce requirements that more accurately reflect the specifications. It thus shows its ability to refine general requirements to make them usable in a given situation. It also showed its ability to support intertwining and specification reuse by showing how requirement satisfaction changes as a specification changes.

When asked whether requirements change, both experts responded that changes occur. GIRAFFE provides support for those changes by analyzing requirements

transformation. The domain expert also noted the difficulty of anticipating and analyzing interactions between specification components. GIRAFFE helps analyze interactions between specification components by stating the effects of specification changes on the requirements.

### Generality of the Implementation

GIRAFFE relies on a domain model to transform requirements. In the preceding section I discussed GIRAFFE's analysis in the domain of on-line registration. In this section I discuss its ability to analyze and transform requirements in other domains. First I describe the kind of domain model that GIRAFFE requires by stating constraints on the model. Then I consider the domain-dependence of elements in GIRAFFE's knowledge base. Finally I discuss the characteristics of domains where GIRAFFE is most effective and characteristics of domains where it is less effective.

### Constraints on the Domain Model

GIRAFFE requires a domain model that uses OPIE's (Anderson, 1993) event representation with several kinds of additional information. One additional kind of information that GIRAFFE uses is a categorization of actions by agent type. Each event must be categorized as being an action provided and controlled by the environment (*env*), an action provided by the artifact but controlled by the environment (*env\_agent*) or an action provided and controlled by the artifact (*art\_agent*). GIRAFFE uses this categorization to decide which actions must be constrained in the requirements.

TABLE 6. Requirements transformations and their domain-dependence

| Name             | Type   | Description  | Domain Dependence  |
|------------------|--|--|--|
| X_add_inits      | transformation                               | add conditions to the initial state                                    | None.  |
| X_add_disj       | transformation                               | make a disjunction in the initial state                                | None.  |
| X_remove_disj    | transformation                               | remove a disjunction from the initial state                            | None.  |
| X_add_cirp       | transformation                               | adds conditions to the common list and removes them from a disjunction | None.  |
| X_remove_inits   | transformation                               | remove conditions from the initial state                               | None.  |
| X_add_pc         | transformation                               | add a constraint on an action  | GIRAFFE must be able to recognize actions that are under the control of agents in the environment.   |
| X_add_abs_pc     | transformation                               | add a constraint on an abstract (more general) action                  | GIRAFFE requires an abstraction hierarchy showing the generality of action types in the domain, plus the same information used for X_add_pc. |
| X_add_and_pc     | transformation                               | add a constraint on a set of actions                                   | GIRAFFE must be able to recognize actions that are under the control of agents in the environment.   |
| X_weaken_ea_pc   | transformation                               | weaken the constraints on environmental agents                         | GIRAFFE must be able to recognize actions that are under the control of agents in the environment.   |
| X_add_etyp_to_ea | transformation                               | add an action to an environmental agent constraint                     | GIRAFFE must be able to recognize actions that are under the control of agents in the environment.   |
| X_obj_gen        | transformation                               | change the generality of an object type                                | GIRAFFE requires an abstraction hierarchy showing the generality of object types in the domain.  |
| derive_privacy   | transformation to derive related requirement | derive new privacy requirements  | GIRAFFE must be able to recognize subtypes of the knows relation, the has_info relation and the info object type.                            |

Table 6 (Continued)

| Name                    | Type   | Description  | Domain Dependence  |
|-------------------------|--|--|--|
| derive_plan_support     | transformation to derive related requirement | derive requirements to achieve initial conditions                      | None.  |
| derive_plan_obstruction | transformation to derive related requirement | derive requirements to prevent initial conditions of safety violations | Relations must be named in such a way that GIRAFFE can recognize negated conditions. |
| derive_failure          | transformation to derive related requirement | derive requirements to prevent failure of achievement requirements     | Relations must be named in such a way that GIRAFFE can recognize negated conditions. |
| derive_repair           | transformation to derive related requirement | derive repair requirements for safety violations                       | Relations must be named in such a way that GIRAFFE can recognize negated conditions. |

For example, the action of a person walking to the registrar's office is provided by and controlled by the environment and so belongs to the `env` category. The action of a person requesting a transcript display is in the `env_agent` category because it is provided by the artifact but controlled by the environment. In other words, without the artifact there would be no way for a student to request a display, but the student decides when and if she will execute the action. Displaying the transcript is an `art_agent` category because a computer program that is part of the artifact controls the action. In addition to the categorization of actions by agent, GIRAFFE requires some kinds of objects and conditions to be represented in a standard way. GIRAFFE can only derive privacy requirements when it finds conditions of a certain form. The form is that the condition must be a subtype of a general `has_info` relation, the agent must be a subtype of the `person` object type, and one other object in the condition must be a subtype of the `info` object type.

TABLE 7. Rating functions and their domain-dependence

| Name                 | Type                   | Description  | Domain Dependence   |
|----------------------|------------------------|--|---|
| G_etyp_xfs           | global rating function | increase rating for common constraints                           | None.   |
| G_new_ea_conflicts   | global rating function | decrease rating for actions constrained in other requirements    | None.   |
| G_not_etyp_reqs      | global rating function | decrease rating of constraints on required actions               | None.   |
| G_not_etyp_xfs       | global rating function | decrease rating of constraints on suggested actions              | None.   |
| R_duration           | local rating function  | award or penalize based on duration of scenarios                 | None.   |
| R_disj_dur           | local rating function  | award or penalize based on duration of new scenarios             | None.   |
| R_triv_disj          | local rating function  | penalize trivial disjunctions                                    | None.   |
| R_satisfaction       | local rating function  | increase ratings of xforms that produce a satisfied requirement. | None.   |
| R_pc_generality      | local rating function  | give AND path constraints higher ratings                         | None.   |
| R_ea_etyps           | local rating function  | decrease ratings of xforms that require more actions             | None.   |
| R_scenarios          | local rating function  | award or penalize based on number of scenarios                   | None.   |
| R_number_of_agents   | local rating function  | award or penalize based on number of scenarios                   | None.   |
| R_pc_motivated_agent | local rating function  | increase ratings if agent is motivated                           | GIRAFFE must be able to recognize subtypes of the knows relation, the has_info relation and the info object type. |

Table 7 (Continued)

| Name             | Type                  | Description  | Domain Dependence   |
|------------------|-----------------------|--|---|
| R_effectiveness  | local rating function | award or penalize based on effectiveness of actions in scenarios | GIRAFFE requires actions in the domain model to have effectiveness attributes where relevant.   |
| R_likelihood     | local rating function | award or penalize based on likelihood of scenarios               | GIRAFFE requires actions in the domain model to have likelihood attributes where relevant.      |
| R_obj_generality | local rating function | award or penalize based on generality of object types            | GIRAFFE requires an abstraction hierarchy showing the generality of object types in the domain. |
| R_init_pers      | local rating function | award or penalize based on number of initial conditions          | Uses domain-specific and domain-independent information.  |

OPIE, as currently implemented, does not support negated conditions. GIRAFFE uses a convention of appending `not_` to the condition name to represent its negation. Its notion of negation is simplistic but useful for analyzing several kinds of requirements such as repair and plan-obstruction requirements.

GIRAFFE uses domain-dependent attributes of actions and conditions in transformations and rating functions. For example, it uses a likelihood attribute for qualitative comparisons of different actions where one person guesses another's password. Attributes of actions and conditions are reflected in scenario and requirement attributes, so without a representation of those attributes GIRAFFE would not be able to reason about transformations that affect the attributes of requirements.

Because of the constraints on the kind of domain model can use, creating a new domain model is a large investment. However, a domain model can be used for more than one problem in the same domain, thus amortizing the cost of creating the domain model. Furthermore, the model can be used for other kinds of analysis, such as those that Anderson (1993) describes, where OPIE uses such a model to suggest specification



changes. Although GIRAFFE's domain model is slightly different than OPIE's, as discussed in the preceding paragraphs, the differences are relatively minor.

### Domain-Independent Transformations and Rating Functions

As stated in a previous section, some of GIRAFFE's transformations and rating functions rely on information specific to a certain domain. However, many of them are domain-independent or require information that is relevant in many domains. Table 6 lists GIRAFFE's transformations and their domain dependence, if any, and Table 7 lists the same kind of information for GIRAFFE's rating functions.

### Other Domains

Although the application discussed in this chapter is from a single domain, on-line registration, GIRAFFE can use the same transformations and rating functions in other domains. GIRAFFE's domain-independent transformations apply to any domain, and many of its other transformations will apply to related domains.

For example, GIRAFFE's derivation of privacy requirements depends on a standard representation of some objects and conditions, but that representation is sufficiently general to be useful in other domains. As one related domain, consider the domain of a travel reservation system. If there is a requirement for customers to be able to find out itinerary information, GIRAFFE can derive a privacy requirement from that requirement. It can also derive and analyze requirements for changes in itineraries or advice on itineraries in the same way that it analyzes changes and advice on academic schedules.

GIRAFFE is most effective in domains where an artifact interacts with agents in its environment. For instance, compilers or numerical analysis programs that are

computationally intensive are artifacts with relatively little interaction and GIRAFFE would thus be less effective for domains with that type of artifact.

### Summary

GIRAFFE's knowledge base is mostly domain-independent. However, GIRAFFE places several constraints on the type of domain model it must have to analyze requirements in a given domain. GIRAFFE is most effective in domains characterized by interaction rather than domains where a single agent does its work with little interaction with its environment.

### Evaluation of the Method

The method embodied in the GIRAFFE program is based on the requirements relations defined in Chapter III. In many cases two requirements (or requirement sets) can be incomparable with respect to those requirements relations. However, the implementation and evaluation of the program show that incomparable requirements do not prevent GIRAFFE from fulfilling its purpose.

GIRAFFE can strengthen and weaken requirements to show how reuse of specification elements changes satisfaction of requirements. It can also address intertwining by directly showing how specification changes affect requirements. GIRAFFE transformed an initial set of general requirements to fit three different specifications, showing its ability to refine requirements. Thus the abilities of the implementation show the value of the method.

Although GIRAFFE achieves its purpose in many ways, it is also limited. Some limitations of the program are inherent to the implementation rather than the general method. In this section I discuss some of the program's limitations and the degree to which they are limitations of the implementation or the general method.

### Limitations in Representation of the Domain

Some limitations in representing a domain are due to a lack of expressiveness in GIRAFFE's current implementation. Two cases of this are negated conditions and distinguishing objects, as described in the following paragraphs:

GIRAFFE and OPIE's representation provides little support for prohibited conditions, or conditions that prevent execution of an action. For example, it would be useful to describe a `tell` action which only occurs if the receiver doesn't already know the information being told. Representing the action in this way prevents the planner from finding plans with innumerable sequences of `tell` actions, all but one of which are superfluous.

Another way of preventing the problem with `tell` described in the previous paragraph would be to provide better ways to distinguish objects in actions, but unfortunately GIRAFFE and OPIE's representation doesn't currently address this problem. For instance, OPIE finds plans with nonsensical actions such as when a student tells something to herself or goes from one place to the same place. (Such actions are nonsensical not because they could not occur, but because they don't represent interesting changes of state at the level of the model used by GIRAFFE.)

Other limitations in representing a domain are due to the inherent difficulty of formally representing an informal system and are not particular to a certain implementation of GIRAFFE. The general problems of knowledge representation and domain analysis are apparent in the difficulty of creating GIRAFFE's domain models.

### Limitations in Finding Scenarios

Some complex scenarios are important but difficult for OPIE to find using planning methods because of the worst-case exponential nature of planning. For example, in the

computer-security domain, one trojan-horse scenario required over a thousand of OPIE's planning cycles to discover, whereas GIRAFFE normally allows two or three hundred cycles. The current implementation of GIRAFFE has no way of discovering such scenarios since it relies on OPIE to find scenarios. However, the same transformations and rating functions that apply to the plans OPIE finds can also apply to more complex scenarios obtained from other sources, so this is a limitation of the implementation rather than the method.

### Compound Specification Changes

GIRAFFE can analyze a specification without taking a specification change into account, and it can analyze addition or removal of an action from the specification. However, the current version is not able to use information from complex specification changes to analyze requirements transformations. For instance, GIRAFFE can analyze Artifact A and Artifact B and it can analyze changes to either specification but it cannot analyze a complex change that adds and deletes operators to change Artifact A to Artifact B in a single change.

### Summary

GIRAFFE achieves its purpose when it finds the strongest requirement satisfied by a specification. In this chapter I evaluated GIRAFFE by showing when it achieved that purpose and when it did not. I summarized its analysis of three specifications and compared GIRAFFE's analysis to that of an analyst and a domain expert.

I also evaluated GIRAFFE as a general method by considering its applicability to other domains. GIRAFFE's knowledge base of transformations and rating functions is largely domain-independent, allowing it to analyze specifications and requirements in other domains, provided that it has a domain model.

The method is limited by availability of a model for the given domain. Some limitations of expressiveness are due to the particular implementation and others are due to the general difficulty of domain analysis and knowledge representation. Despite these limitations, GIRAFFE achieves its purpose. In the next chapter I summarize GIRAFFE's abilities and the contributions of this research. I also describe future work which addresses some of the limitations discussed in this chapter and considers further research in requirements transformation.

## CHAPTER VII

### CONCLUSION

#### Introduction

In the final chapter of this dissertation I summarize GIRAFFE's contributions and limitations. In describing GIRAFFE's limitations I suggest ways of extending GIRAFFE to address some of those limitations as possible future work.

#### Contributions

The main contribution of this work is the definition of the IS-STRONGER-THAN relation and its incorporation into GIRAFFE's knowledge base as a set of transformations and rating functions. GIRAFFE's knowledge base includes transformations that strengthen and weaken requirements by adding and removing qualifications and by generalizing or specializing object types. GIRAFFE can also change requirements by modifying values of attributes, although it does not use exact values for most attributes.

GIRAFFE can use its knowledge base to incrementally strengthen or weaken requirements and state the strongest requirement satisfied by a specification. GIRAFFE's ability to stretch requirements in this way helps the client avoid the problems of unsatisfied, impractical requirements and unnecessary weakening. In this research, I have applied work in dimensions to the problem of requirements engineering and extended it to include qualifications stated in terms of actions and conditions in addition to sets of attributes and values used in other work. GIRAFFE also use object type generality in comparing and transforming requirements.

GIRAFFE's transformations allow it to suggest new requirements derived from previously stated requirements. By applying planning research in goal relations to requirements engineering, I have developed a set of transformations to derive plan-support and plan-obstruction requirements, repair requirements, failure requirements and privacy requirements.

In addition to GIRAFFE's knowledge base, another contribution of this work is GIRAFFE's use of general scenarios and its method of finding them. Rather than finding a single plan that achieves a stated goal from a completely specified initial state, GIRAFFE's planner (a modified version of the OPIE planner called MOPIE) can find multiple plans from a partially specified initial state. Using OPIE's constraint propagation features, MOPIE can find plans with more specific object types, and by generalizing object types before planning begins it can find plans with more general object types. Thus MOPIE can find plans that are generalized in three respects: object types, multiple paths and initial state.

These contributions provide the means for improving the requirements engineering process by using requirements transformation to address intertwining, improve evaluation, facilitate reuse of specification components, and refine general requirements to make them applicable for a given situation.

### Limitations and Future Work

In this section I summarize some limitations of GIRAFFE and suggest ways of addressing those limitations in future work. I discuss the difficulty of constructing domain models, output formats, and problems in finding scenarios, along with future work that relates to each issue.

### Constructing Domain Models

One limitation of GIRAFFE is the difficulty of constructing domain models. The two factors that cause the most difficulty are the formal representation and the size. Because of the inherent difficulty of formally modelling a system, GIRAFFE (and every similar program) will always be somewhat limited but there are ways of addressing this problem to some degree. Two ways to address this problem are creating a more expressive language for the model and providing automated support for its development.

One possibility for supporting construction of domain models is to help at a syntactic level. Automation could help with bookkeeping by checking syntax, summarizing various attributes of the model, and so on.

A more sophisticated tool could use knowledge acquisition techniques to help build the model. Using repertory grids (Boose, 1986) is one common technique that is well-suited to defining GIRAFFE's *describe* operators. An easy-to-use, automated tool would allow the *describe* operators to be changed to fit each problem situation more closely.

Knowledge acquisition techniques that use repertory grids produce rules that can be used to classify instances. So, for instance, the set of rules might classify (within some uncertainty parameter) a student who is taking lower division courses, not working on a thesis, and living on campus as an undergraduate. The attributes and classes are defined during the acquisition process and then rules are generated from the acquired grid of values.

GIRAFFE would use grids to produce operators that describe instances. Given an undergraduate, it could describe the student as taking lower division courses, etc. It would define subclasses as necessary to account for attributes that do not apply to an entire group, such as living on campus. Since there would be many subclasses, GIRAFFE's abilities to



generalize and specialize scenarios would be important. Some scenarios might be possible only for students living on campus (perhaps if they have easy access to university terminals and computers), and some possible for any undergraduate.

A more expressive representation language would also alleviate some of the difficulty of constructing GIRAFFE's domain models. Rather than stating all actions in MOPIE's STRIPS-like language, one could use more general descriptions of actions than listing preconditions and effects.

### Output Format

GIRAFFE is further limited in that its ability to describe requirements and scenarios is limited to its own formal representation. The difficulty of reading output from GIRAFFE makes it unsuitable for use by a client or even a typical analyst. Output in other forms, such as text and diagrams, would allow more people to use it with less training. Because of the large volume of many software engineering documents, a database form that could be summarized and queried, or a hypertext format, could provide useful alternatives to flat text.

### Finding Scenarios

GIRAFFE is also limited in that it finds too many and too few scenarios. As discussed in Chapter V, MOPIE can't find some scenarios because they are too complex. Other scenarios require actions that are difficult to describe in terms of simple preconditions and effects. In Chapter V I looked at three possible methods for extending GIRAFFE's scenario-finding abilities, including forward chaining, acquisition, and case-based planning.

GIRAFFE finds too many scenarios when MOPIE spends its time finding trivial variations of scenarios. Since there can be infinitely many plans for some problems, and

since GIRAFFE's plan generalization methods lead to even more plans, MOPIE must constrain the time that it spends looking for scenarios. If MOPIE could more effectively recognize which differences in plans are trivial it could spend more time on plans that are more important for GIRAFFE's purposes.

The current version of GIRAFFE has some ways to push constraints into the planner, but a better facility for doing so would improve MOPIE's abilities in discriminating between significant and insignificant differences in plans. GIRAFFE can state path constraints in terms of abstract action types, but those constraints are applied as filters and MOPIE ignores them when looking for scenarios. If MOPIE had the ability to use constraints on abstract actions and other constraints during planning, it would find fewer irrelevant scenarios.

### Conclusion

In this dissertation I have presented a method of requirements transformation that is implemented in the GIRAFFE program. GIRAFFE supports intertwining in requirements engineering, helps evaluate specifications, promotes reuse of specification components and allows refinement of general requirements.

GIRAFFE's knowledge base comprises transformations and rating functions that provide a way to incrementally weaken or strengthen requirements, thus allowing it to find the strongest requirement satisfied by a specification. GIRAFFE's knowledge base also allows it to derive new requirements from existing ones using transformations based on goal relations.

GIRAFFE's limitations include its inability to find some kinds of scenarios, the difficulty of constructing domain models, and the lack of an easy-to-understand output format. Future work to address those limitations includes additional methods of finding

scenarios, automated support for construction of domain models along with a more expressive representation language, and output in text, diagrams or hypertext.

## APPENDIX A

## GLOSSARY

**action**—An action is an event caused by an agent that changes conditions, for example, a student adding a class to her schedule. Formally, an *action* is a tuple  $\langle \textit{capability}, \textit{agent}, \textit{objects}, \textit{consumed}, \textit{used}, \textit{produced} \rangle$  where *capability* is a symbol representing the capability required for the action, *agent* is an object definition, *objects* is a list of object definitions, and *consumed*, *used*, and *produced* are sets of conditions.

**agent**—a person or program capable of executing actions that change the state of the artifact or its environment. Examples are a student or a registration program.

**analyst**—a person or program that assists with requirements engineering but has no authority to change requirements.

**artifact**—a system including software and hardware components. An artifact also includes roles filled by people.

**attribute**—An attribute is a pair  $\langle \textit{attr-name}, \textit{attr-value} \rangle$ , where *attr-name* is a symbol, and *attr-value* is a number or a symbol. Attributes that GIRAFFE uses include importance, likelihood, duration and effectiveness.

**capability**—a specification element representing the means for an agent to change the conditions that hold in a situation. An example is {phone request schedule}. The formal definition of a capability is:

Definition 2

A *capability* is a tuple  $\langle \textit{name}, \textit{agent}, \textit{otyps}, \textit{consumed}, \textit{used}, \textit{produced}, \textit{Super}, \textit{Sub} \rangle$  where *name* is a symbol, *agent* is a symbol representing an object type, *otyps* is a list of symbols representing object types, and *consumed*, *used*, and *produced* are sets of conditions. *Super* and *Subs* are sets of symbols, where each symbol represents an action type, and are part of a hierarchy of action types. □

**client**—a person who can authoritatively state and change requirements for an artifact.

**condition**—a relation that holds between zero or more objects at a time specified relative to a transition or execution of actions in a plan. Formally, a condition is a pair  $\langle \textit{relation}, \textit{objects} \rangle$  where *relation* is a symbol and *objects* is a list of symbols representing the objects that play roles in the relationship.

describe operator—an operator used by GIRAFFE to determine what conditions might plausibly be true in a domain.

final state—the state that holds after a transition occurs.

GIRAFFE—a program that supports requirements transformation. GIRAFFE's knowledge base is derived from the IS-STRONGER-THAN relation and other relations between requirements.

initial state—a state that holds before a transition occurs.

object—an entity in a domain. GIRAFFE uses a symbol, called an object name or *obj\_name*, to represent an object.

object definition—An object definition is a pair  $\langle obj\_type, obj\_name \rangle$ , where *obj\_type* and *obj\_name* are symbols. *Obj\_type* is a symbol representing a type defined in an object-type hierarchy.

operator—the formalism that a planner uses to represent events that cause a change in conditions. GIRAFFE defines operator sets for OPIE (a planning program) using capabilities and describe operators.

path constraint—a statement that a particular action or intermediate condition will not occur in a transition.

qualification—an assumption required to allow an achievement transition to occur or prevent a safety transition from occurring. Qualifications can be stated in terms of initial conditions or in terms of actions that occur in a path. The formal definition of a qualification is:

Definition 1

A *qualification*,  $q$ , is defined as follows:

$$q ::= q_{\text{prim}} \mid q_{\text{complex}}$$

$$q_{\text{prim}} ::= q_{\text{init}} \mid q_{\text{action}}$$

$q_{\text{init}} ::= \langle q_{\text{type}}, \text{condition} \rangle$ , where  $q_{\text{type}}$  is the symbol `has_init`.

$$q_{\text{action}} ::= \langle q_{\text{type}}, \text{action\_type\_name}, \text{agent\_type\_name} \rangle \mid \langle q_{\text{type}}, \text{action\_type\_name} \rangle$$

where  $q_{\text{type}}$  is one of the symbols `has_action`, `does_not_have_action`, or `env_action`.

$$q_{\text{complex}} ::= q_{\text{complex}} \vee q_{\text{term}} \mid q_{\text{term}}$$

$$q_{\text{term}} ::= q_{\text{term}} \wedge q_{\text{prim}} \mid q_{\text{prim}}$$

*Action\_type\_name*, and *agent\_type\_name* are symbols. □

**requirements**—a high-level description of an artifact. Unlike the functional specification, requirements are not stated in terms of capabilities and actions. In the method of requirements transformation described in this dissertation, requirements are state transitions that the artifact should facilitate or prevent. Names of requirements are enclosed in angle brackets, such as `<find out schedule>`.

A requirement is formally defined as a tuple  $\langle \text{name}, \text{type}, O, \text{trans}, Q, A, \text{Related} \rangle$ , where *name* and *type* are symbols, *O* is a set of object definitions, *trans* is a state transition, *Q* is a set of qualifications, and *A* is a set of attributes. *Related* is a set of symbols representing links to related requirements.

**requirements engineering**—the process of stating requirements for an artifact.

**requirements transformation**—a method of requirements engineering defined in this dissertation. In this method, the client states a specification change (or change in functionality) to an analyst, who responds by telling the client the strongest requirement satisfied by the specification after the change in functionality.

**scenario**—a sequence of actions showing how a transition from an initial state to a final state can occur.

**specification**—A description of an artifact in terms of the capabilities it provides.

**state**—a set of conditions that hold at a certain time relative to a transition or to a sequence of actions.

transition—a change from one state of an artifact and its environment to another. Formally, a transition is a tuple  $\langle IC, PC, FC, Scenarios \rangle$ , where  $IC$  and  $FC$  are sets of conditions,  $PC$  is a set of path constraints, and  $Scenarios$  is a set of scenarios.

## APPENDIX B

## DOMAIN MODEL

This appendix gives GIRAFFE's model for the domain of on-line registration. The components of the model are the definition of capabilities in the environment, the set of initial requirements used by GIRAFFE for this domain, and some object definitions.

Definition of Capabilities in the Environment

The first line of each definition is the word `def_etyp`, followed by the name of the capability in quotes. The second line is a description of the capability. The third line, beginning with `:o`, lists the objects referred to in the capability. Objects in capabilities are defined by appending a number to the name of an object type, so `person1` refers to an object of type `person`. The first object in the list is the agent.

The list following the keyword `:-` lists the conditions *consumed* when an agent uses the capability to execute an action. Consumed conditions are required to hold before an action executes and do not hold after the action has executed. Conditions are a list where the first element is a relation name and the second is a list of the objects for that condition.

The list following the keyword `:=` lists the conditions *used* when an agent uses the capability to execute an action. Conditions that are used are required for an action to execute but are still true after the action occurs.

The list following the keyword `:+` lists the conditions *produced* when an agent uses the capability to execute an action. Conditions that are produced are true after the execution of the action.

In the first capability listed, there is one consumed condition and one produced condition. In the consumed condition, the agent is in an arbitrary location, and in the produced condition the agent is at the registrar's office.

The format of `describe` operators is similar to the format of capabilities that represent actions, except that they have no agent.

```

;----- GO SOMEPLACE -----
(def_etyp "go to registrars office"
:d "a person goes to the registrar's office"
:o '(person1 loc1)
:- '((at (person1 loc1)))
:+ '((at_registrars_office (person1)))
:at '((duration :val 1)(category :val env))
)

```



```

(def_etyp "go to advisors office"
:d "a person goes to an advisor's office"
:o '(person1 loc1)
:- '((at (person1 loc1))
      (has_appointment (person1)) )
:+ '((at_advisors_office (person1)))
:at '((duration :val 1)(category :val env))
)

(def_etyp "go to terminal room"
:d "a person goes to a place with a terminal"
:o '(person1 loc1 loc2 terminal1)
:- '((at (person1 loc1)))
:+ '((at_terminal_room (person1))
      (has_term_access (person1 terminal1))
      (can_see_screen (person1 terminal1)) )
:at '((duration :val 1)(category :val env))
)

;----- READ SOMETHING FROM PAPER -----
(def_etyp "read transcript doc"
:d "a person reads transcript info from a transcript"
:o '(person1 transcript_info1 transcript1)
:= '((has_transcript (person1 transcript1))
      (doc_has_t_info (transcript1 transcript_info1)) )
:+ '((knows_t (person1 transcript_info1)))
:at '((category :val env))
)

(def_etyp "read schedule doc"
:d "a person reads schedule info from a schedule listing"
:o '(person1 sched_info1 schedule_listing1)
:= '((has_schedule_listing (person1 schedule_listing1))
      (doc_has_s_info (schedule_listing1 sched_info1)) )
:+ '((knows_s (person1 sched_info1)))
:at '((category :val env))
)

(def_etyp "read catalog doc"
:d "a person reads school and help info from a catalog"
:o '(person1 school_info1 help_info1 catalog1)
:= '((has_catalog (person1 catalog1))
      (doc_has_c_info (catalog1 school_info1))
      (doc_has_h_info (catalog1 help_info1)) )
:+ '((knows_c (person1 school_info1))
      (knows_h (person1 help_info1)) )
:at '((category :val env))
)

```

```

;----- STUDENT INFORMATION CHANGES -----

(def_etyp "info changes"
:d "a person moves, changes name, etc."
:o '(person1 student_info1 student_info2)
:- '((has_student_info (person1 student_info1)))
:= '((diff_info (student_info1 student_info2)))
:+ '((has_student_info (person1 student_info2))
      (not_has_student_info (person1 student_info1)) )
:at '((category :val env)(restrict :val 1))
)

;----- LOSE, GIVE OR STEAL A SCHEDULE LISTING -----

(def_etyp "lose a schedule listing"
:d "a person loses a schedule listing"
:o '(person1 schedule_listing1)
:- '((has_schedule_listing (person1 schedule_listing1)))
:+ '((not_has_schedule_listing (person1 schedule_listing1))
      (lost (person1 schedule_listing1)) )
:at '((category :val env)(restrict :val 1))
)

(def_etyp "steal a schedule listing"
:d "one person steals a schedule listing from another"
:o '(person1 person2 schedule_listing1)
:- '((has_schedule_listing (person2 schedule_listing1)))
:= '((diff (person1 person2)))
:+ '((not_has_schedule_listing (person2 schedule_listing1))
      (has_schedule_listing (person1 schedule_listing1))
      (stolen (person1 schedule_listing1 person2)) )
:at '((category :val env)(restrict :val 1))
)

(def_etyp "give a schedule listing"
:d "one person gives a schedule listing to another"
:o '(person1 person2 schedule_listing1)
:- '((has_schedule_listing (person1 schedule_listing1)))
:= '((diff (person1 person2)))
:+ '((not_has_schedule_listing (person1 schedule_listing1))
      (has_schedule_listing (person2 schedule_listing1))
      (given (person1 schedule_listing1 person2)) )
:at '((category :val env)(restrict :val 1))
)

;----- LOSE, GIVE OR STEAL A CATALOG -----

(def_etyp "lose a schedule listing"
:d "a person loses a schedule listing"
:o '(person1 schedule_listing1)
:- '((has_schedule_listing (person1 schedule_listing1)))
:+ '((not_has_schedule_listing (person1 schedule_listing1))
      (lost (person1 schedule_listing1)) )
:at '((category :val env)(restrict :val 1))
)

```

```

(def_etyp "steal a schedule listing"
:d "one person steals a schedule listing from another"
:o '(person1 person2 schedule_listing1)
:- '((has_schedule_listing (person2 schedule_listing1)))
:= '((diff (person1 person2)))
:+ '((not_has_schedule_listing (person2 schedule_listing1))
      (has_schedule_listing (person1 schedule_listing1))
      (stolen (person1 schedule_listing1 person2)) )
:at '((category :val env)(restrict :val 1))
)

```

```

(def_etyp "give a catalog"
:d "one person gives a catalog to another"
:o '(person1 person2 catalog1)
:- '((has_catalog (person1 catalog1)))
:= '((diff (person1 person2)))
:+ '((not_has_catalog (person1 catalog1))
      (has_catalog (person2 catalog1))
      (given (person1 catalog1 person2)) )
:at '((category :val env)(restrict :val 1))
)

```

;----- LOSE, GIVE OR STEAL AN ID -----

```

(def_etyp "lose an id"
:d "a person loses an id"
:o '(person1 id1)
:- '((has_id (person1 id1)))
:+ '((not_has_id (person1 id1))
      (lost (person1 id1)) )
:at '((category :val env)(restrict :val 1))
)

```

```

(def_etyp "steal an id"
:d "one person steals an id from another"
:o '(person1 person2 id1)
:- '((has_id (person2 id1)))
:= '((diff (person1 person2)))
:+ '((not_has_id (person2 id1))
      (has_id (person1 id1))
      (stolen (person1 id1 person2)) )
:at '((category :val env)(restrict :val 1))
)

```

```

(def_etyp "give an id"
:d "one person gives an id to another"
:o '(person1 person2 id1)
:- '((has_id (person1 id1)))
:= '((diff (person1 person2)))
:+ '((not_has_id (person1 id1))
      (has_id (person2 id1))
      (given (person1 id1 person2)) )
:at '((category :val env)(restrict :val 1))
)

```

```
;----- LOSE, GIVE OR STEAL A TRANSCRIPT -----
```

```
(def_etyp "lose a transcript"
:d "a person loses a transcript"
:o '(person1 transcript1)
:- '((has_transcript (person1 transcript1)))
:+ '((not_has_transcript (person1 transcript1))
      (lost (person1 transcript1)) )
:at '((category :val env)(restrict :val 1))
)
```

```
(def_etyp "steal a transcript"
:d "one person steals a transcript from another"
:o '(person1 person2 transcript1)
:- '((has_transcript (person2 transcript1)))
:= '((diff (person1 person2)))
:+ '((not_has_transcript (person2 transcript1))
      (has_transcript (person1 transcript1))
      (stolen (person1 transcript1 person2)) )
:at '((category :val env)(restrict :val 1))
)
```

```
(def_etyp "give a transcript"
:d "one person gives a transcript to another"
:o '(person1 person2 transcript1)
:- '((has_transcript (person1 transcript1)))
:= '((diff (person1 person2)))
:+ '((not_has_transcript (person1 transcript1))
      (has_transcript (person2 transcript1))
      (given (person1 transcript1 person2)) )
:at '((category :val env)(restrict :val 1))
)
```

```
;----- LOSE, GIVE OR STEAL A CATALOG -----
```

```
(def_etyp "lose a catalog"
:d "a person loses a catalog"
:o '(person1 catalog1)
:- '((has_catalog (person1 catalog1)))
:+ '((not_has_catalog (person1 catalog1))
      (lost (person1 catalog1)) )
:at '((category :val env)(restrict :val 1))
)
```

```
(def_etyp "steal a catalog"
:d "one person steals a catalog from another"
:o '(person1 person2 catalog1)
:- '((has_catalog (person2 catalog1)))
:= '((diff (person1 person2)))
:+ '((not_has_catalog (person2 catalog1))
      (has_catalog (person1 catalog1))
      (stolen (person1 catalog1 person2)) )
:at '((category :val env)(restrict :val 1))
)
```

```

(def_ety "give a catalog"
:d "one person gives a catalog to another"
:o '(person1 person2 catalog1)
:- '((has_catalog (person1 catalog1)))
:= '((diff (person1 person2)))
:+ '((not_has_catalog (person1 catalog1))
      (has_catalog (person2 catalog1))
      (given (person1 catalog1 person2)) )
:at '((category :val env)(restrict :val 1))
)

;----- READ SOMETHING FROM A SCREEN -----
(def_ety "read transcript info"
:d "read a person's transcript info from screen"
:o '(person1 transcript_infol terminal1)
:= '((t_on_screen (transcript_infol terminal1))
      (can_see_screen (person1 terminal1)) )
:+ '((knows_t (person1 transcript_infol)))
:at '((category :val env))
)

(def_ety "read schedule info"
:d "read a person's schedule info from screen"
:o '(person1 sched_infol terminal1)
:= '((s_on_screen (sched_infol terminal1))
      (can_see_screen (person1 terminal1)) )
:+ '((knows_s (person1 sched_infol)))
:at '((category :val env))
)

(def_ety "read rec sched"
:d "see a student's recommended schedule for a term"
:o '(person1 rec_sched1 terminal1)
:= '((r_on_screen (rec_sched1 terminal1))
      (can_see_screen (person1 terminal1)))
:+ '((knows_r (person1 rec_sched1)))
:at '((category :val env))
)

(def_ety "read school info"
:d "read school information from a terminal screen"
:o '(person1 school_infol terminal1)
:= '((c_on_screen (school_infol terminal1))
      (can_see_screen (person1 terminal1)) )
:+ '((knows_c (person1 school_infol)))
:at '((category :val env))
)

```

```

(def_ety "read interrupt help info"
:d "read help information from a terminal screen"
:o '(person1 help_info1 line_terminal1)
:= '((h_on_screen (help_info1 line_terminal1))
      (can_see_screen (person1 line_terminal1)) )
:+ '((knows_h (person1 help_info1))
      (_is_+read_line_help) )
:at '((category :val env))
)

(def_ety "read concurrent help info"
:d "read help information from a screen of a window terminal"
:o '(person1 help_info1 window_terminal1)
:= '((h_on_screen (help_info1 window_terminal1))
      (can_see_screen (person1 window_terminal1)) )
:+ '((knows_h (person1 help_info1))
      (_is_+read_window_help) )
:at '((category :val env)(effectiveness :val 1))
)

;----- TELL SOMEONE -----
(def_ety "tell transcript info"
:d "tell someone else information"
:o '(person1 person2 transcript_info1)
:= '((knows_t (person1 transcript_info1))
      (diff (person1 person2)) )
:+ '((knows_t (person2 transcript_info1)))
:at '((category :val env))
)

(def_ety "tell pac"
:d "tell someone else information"
:o '(person1 person2 pac1)
:= '((knows_p (person1 pac1))
      (diff (person1 person2)) )
:+ '((knows_p (person2 pac1)))
:at '((category :val env))
)

;----- GUESS PAC -----
(def_ety "guess easy PAC"
:d "person guesses an easy-to-guess PAC"
:o '(person2 person1 pac1)
:= '((has_pac (person1 pac1))
      (is_easy_pac (pac1)) )
:+ '((knows_p (person2 pac1))
      (_is_+guess_easy_pac) )
:at '((likelihood :val 1)(category :val env))
)

```

```

(def_etyp "guess hard PAC"
:d "person guesses a hard-to-guess PAC"
:o '(person2 person1 pac1)
:= '((has_pac (person1 pac1))
      (is_hard_pac (pac1)) )
:+ '((knows_p (person2 pac1))
      (_is_+guess_hard_pac) )
:at '((likelihood :val -1)(category :val env))
)

;----- FORGET PAC -----

(def_etyp "forget easy PAC"
:d "person forgets an easy-to-guess PAC"
:o '(person1 pac1)
:= '((has_pac (person1 pac1))
      (is_easy_pac (pac1))
      (knows_p (person1 pac1)) )
:+ '((not_knows_p (person1 pac1))
      (_is_+forget_easy_pac) )
:at '((likelihood :val -1)(category :val env)(restrict :val 1))
)

(def_etyp "forget hard PAC"
:d "person forgets a hard-to-guess PAC"
:o '(person1 pac1)
:= '((has_pac (person1 pac1))
      (is_hard_pac (pac1))
      (knows_p (person1 pac1)) )
:+ '((not_knows_p (person1 pac1))
      (_is_+forget_hard_pac) )
:at '((likelihood :val 1)(category :val env)(restrict :val 1))
)

;----- DESCRIBE THINGS -----

(def_etyp "describe person"
:d "describe a person"
:o '(person1 terminal1 loc1)
:- '((is_person (person1)))
:= '((is_anyplace (loc1)))
:+ '((is_+person (person1))
      (at (person1 loc1))
      (has_money (person1))
      (has_phone_access (person1))
      (has_term_access (person1 terminal1))
      (can_see_screen (person1 terminal1)) )
:at '((category :val describe))
)

```

```

(def_etyp "describe student w/ schedule"
:d "describe a student (has transcript, schedule, etc.)"
:o '(person1 transcript_info1 pac1 degree1 term1 rec_sched1 terminal1
sched_info1 loc1
    id1 student_info1)
:- '((_is_person (person1))
    (_is_transcript_info (transcript_info1))
    (_is_id (id1))
    (_is_student_info (student_info1))
    (_is_sched_info (sched_info1)) )
:= '((_is_terminal (terminal1))
    (_is_degree (degree1))
    (_is_term (term1))
    (_is_pac (pac1))
    (_is_anyplace (loc1))
    (_is_rec_sched (rec_sched1)) )
:+ '((_is_+student (person1))
    (has_transcript_info (person1 transcript_info1))
    (has_schedule_info (person1 term1 sched_info1))
    (not_sent_schedule (person1))
    (has_rec_sched (person1 degree1 term1 rec_sched1))
    (has_student_info (person1 student_info1))
    (is_ID_for (id1 person1))
    (has_id (person1 id1))
    (has_pac (person1 pac1))
    (_is_uknwn_diff (pac1))
    (knows_p (person1 pac1))
    (knows_t (person1 transcript_info1))
    (needs_advisor (person1))
    (has_money (person1))
    (at (person1 loc1))
    (has_phone_access (person1))
    (has_term_access (person1 terminal1))
    (can_see_screen (person1 terminal1)) )
:at '((category :val describe))
)

(def_etyp "describe advisor"
:d "describe an advisor"
:o '(person1 database1)
:- '((_is_advisor (person1)))
:= '((_is_database (database1)))
:+ '((_is_+advisor (person1))
    (can_advise (person1))
    (has_time_available (person1))
    (person_has_access_to (person1 database1)) )
:at '((category :val describe))
)

```



```

(def_etyp "describe registrar worker"
:d "describe an advisor"
:o '(person1 databasel)
:- '((_is_registrar (person1)))
:= '((_is_database (databasel)))
:+ '((_is_+registrar (person1)
      (works_at_registrars (person1))
      (person_has_access_to (person1 databasel)) )
:at '((category :val describe))
)

(def_etyp "describe phone prog"
:d "describe a phone access program"
:o '(program1 databasel)
:- '((_is_program (program1)))
:+ '((has_access_to (program1 databasel))
      (connection_available (program1))
      (can_do_phone_IO (program1)) )
:at '((category :val describe))
)

(def_etyp "describe database"
:d "describe a database"
:o '(databasel transcript_info1 sched_info1 student_info1)
:- '((_is_database (databasel)))
; := '((_is_transcript_info (transcript_info1))
;      (_is_sched_info (sched_info1))
;      (_is_student_info (student_info1)) )
:+ '((db_has_t_in (databasel transcript_info1))
      (db_has_s_in (databasel sched_info1))
      (db_has_si_in (databasel student_info1)) )
:at '((category :val describe))
)

(def_etyp "describe class"
:d "describe a class"
:o '(class1 seat1 term1)
:- '((_is_class (class1))
      (_is_seat (seat1)) )
:+ '((seat_available (seat1 class1 term1)))
:at '((category :val describe))
)

(def_etyp "describe system"
:d "describe a program"
:o '(program1 databasel terminal1 loc1)
:- '((_is_program (program1)))
:= '((_is_terminal (terminal1)))
:+ '((has_access_to (program1 databasel))
      (connection_available (program1))
      (can_do_terminal_IO (program1 terminal1))
      (not_in_use (terminal1))
      (terminal_at (terminal1 loc1)) )
:at '((category :val describe))
)

```

```
;(def_etyp "describe line-terminal system"
;d "describe a program"
;o '(program1 database1 line_terminal1 loc1)
;- '((_is_program (program1)))
;:= '((_is_terminal (line_terminal1)))
;+ '((has_access_to (program1 database1))
;    (connection_available (program1))
;    (can_do_terminal_IO (program1 line_terminal1))
;    (not_in_use (line_terminal1))
;    (terminal_at (line_terminal1 loc1)) )
;:at '((category :val describe))
;)

(def_etyp "describe help info"
;d "describe help info of a program"
;o '(program1 help_info1)
;- '((_is_help_info (help_info1)))
;:= '((_is_program (program1)))
;+ '((has_help_info (program1 help_info1)))
;:at '((category :val describe))
)
```

### Initial Requirement Set

The initial requirements are listed here in the format that GIRAFFE uses to read requirements from a file. The first line of the requirement definition is the word `def_req` followed by the name of the requirement. The second line is a description of the requirement, which GIRAFFE uses when describing transformed requirements.

The third line is the type of the requirement, where `'good` is used to represent achievement requirements—desired transitions, and `'bad` is used to represent safety requirement—prohibited transitions.

A list of object definitions follows the keyword `:o`, where each object definition is the name of an object followed by the name of an object type. Objects in the requirements represent arbitrary members of a class. All of the definitions listed here use a similar object list, although not every definition uses every object in the list.

The lists following the keywords `:i` and `:g` (for "goal") define the initial and final states, respectively, of the requirement's transition. In many cases the initial state is empty; in those cases, all of the initial conditions used in a scenario are produced by `describe` operators.

Some requirements have a keyword `:cycles`, which is an indication of how much time the planner should spend looking for scenarios for the requirement.

The first requirement is an achievement requirement with one condition, that Basil wants to take the class CIS121 in the Winter 93 term, in the initial state. There are two conditions in the final state, which represent a state where Basil has the class in his schedule.

```

;----- REQUIREMENTS -----
(def_req "add_class"
:d "A student should be able to add a class."
:r 'good
:o '(
  (abigail person) (basil person)
  (registrar_a registrar) (advisor_a advisor)
  (cis121 class) (121seat seat)
  (basils_trans_info transcript_info)
  (basils_sched_info sched_info)
  (basils_id id) (basils_transcript transcript)
  (basils_st_info student_info)
  (basils_pac pac)
  (win93 term) (CIS_BS degree)
  (basils_rs rec_sched)
  (online_reg program) (reg_db database)
  (terminal_a terminal)
  (registrars_office loc) (anyplace loc)
)
:i '(
(wants (basil cis121 win93))
)
:g '(
(has_schedule_info (basil win93 basils_sched_info))
(class_in_sched (cis121 basils_sched_info))
))

```

```
-----  
(def_req "find_out_schedule"  
:d "A student should be able to find out his/her schedule information."  
:r 'good  
:o '(  
  (abigail person)(basil person)  
  (registrar_a registrar)(advisor_a advisor)  
  (basils_trans_info transcript_info)  
  (basils_sched_info sched_info)  
  (basils_id id)(basils_transcript transcript)  
  (basils_schedule_listing schedule_listing)  
  (basils_st_info student_info)  
  (basils_pac pac)  
  (win93 term)(CIS_BS degree)  
  (the_school_info school_info)(reg_help_info help_info)  
  (basils_rs rec_sched)  
  (online_reg program)(reg_db database)  
  (terminal_a terminal)  
  (registrars_office loc)(anyplace loc)  
)  
:i '(  
)  
:g '(  
(has_schedule_info (basil win93 basils_sched_info))  
(knows_s (basil basils_sched_info))  
)  
)  
-----
```

```

(def_req "unwanted_class"
:d "A student should not have an unwanted class."
:r 'bad
:o '(
  (abigail person)(basil person)
  (registrar_a registrar)(advisor_a advisor)
  (cis121 class)(121seat seat)
  (basils_trans_info transcript_info)
  (basils_sched_info sched_info)
  (basils_id id)(basils_transcript transcript)
  (basils_st_info student_info)
  (basils_pac pac)
  (win93 term)(CIS_BS degree)
  (basils_rs rec_sched)
  (online_reg program)(reg_db database)
  (terminal_a terminal)
  (registrars_office loc)(anyplace loc)
)
:i '(
(not_wants (basil cis121 win93))
)
:g '(
(has_schedule_info (basil win93 basils_sched_info))
(class_in_sched (cis121 basils_sched_info))
))

;-----

(def_req "get_advice"
:d "A student should be able to find out his/her recommended schedule."
:r 'good
:o '(
  (abigail person)(basil person)
  (registrar_a registrar)(advisor_a advisor)
  (basils_trans_info transcript_info)
  (basils_sched_info sched_info)
  (basils_id id)(basils_transcript transcript)
  (basils_st_info student_info)
  (basils_pac pac)
  (win93 term)(CIS_BS degree)
  (basils_rs rec_sched)
  (online_reg program)(reg_db database)
  (terminal_a terminal)
  (registrars_office loc)(anyplace loc)
)
:i '(
)
:g '(
(has_rec_sched (basil CIS_BS win93 basils_rs))
(knows_r (basil basils_rs))
))

;-----

```

```

(def_req "get_transcript"
:d "A student should be able to get a copy of his/her transcript"
:r 'good
:o '(
  (abigail person)(basil person)
  (registrar_a registrar)(advisor_a advisor)
  (basils_trans_info transcript_info)
  (basils_sched_info sched_info)
  (basils_id id)(basils_transcript transcript)
  (basils_st_info student_info)
  (basils_pac pac)
  (win93 term)(CIS_BS degree)
  (basils_rs rec_sched)
  (online_reg program)(reg_db database)
  (terminal_a terminal)
  (registrars_office loc)(anyplace loc)
)
:i '(
)
:g '(
(has_transcript (basil basils_transcript))
(doc_has_t_info (basils_transcript basils_trans_info))
))

;-----

(def_req "get_transcript_info"
:d "A student should be able to find out his/her transcript
information."
:r 'good
:o '(
  (abigail person)(basil person)
  (registrar_a registrar)(advisor_a advisor)
  (basils_trans_info transcript_info)
  (basils_sched_info sched_info)
  (basils_id id)(basils_transcript transcript)
  (basils_st_info student_info)
  (basils_pac pac)
  (win93 term)(CIS_BS degree)
  (basils_rs rec_sched)
  (online_reg program)(reg_db database)
  (terminal_a terminal)
  (registrars_office loc)(anyplace loc)
)
:i '()
:g '(
(has_transcript_info (basil basils_trans_info))
(knows_t (basil basils_trans_info))
)
::pc '(
(not_per (at (basil registrars_office)))
;)
)

;-----

```

```

(def_req "get_school_info"
:d "A student should be able to find out information about his/her
school"
:r 'good
:o '(
  (abigail person) (basil person)
  (registrar_a registrar) (advisor_a advisor)
  (basils_trans_info transcript_info)
  (basils_sched_info sched_info)
  (basils_id id) (basils_transcript transcript)
  (basils_st_info student_info)
  (catalog_a catalog)
  (basils_pac pac)
  (win93 term) (CIS_BS degree)
  (the_school_info school_info) (reg_help_info help_info)
  (basils_rs rec_sched)
  (online_reg program) (reg_db database)
  (terminal_a terminal)
  (registrars_office loc) (anyplace loc)
)
:i '()
:g '(
(knows_c (basil the_school_info))
))

```

```

;-----
(def_req "get_help_info"
:d "A student should be able to find out help information"
:r 'good
:o '(
  (abigail person) (basil person)
  (registrar_a registrar) (advisor_a advisor)
  (basils_trans_info transcript_info)
  (basils_sched_info sched_info)
  (basils_id id) (basils_transcript transcript)
  (basils_st_info student_info)
  (catalog_a catalog)
  (basils_pac pac)
  (win93 term) (CIS_BS degree)
  (the_school_info school_info) (reg_help_info help_info)
  (basils_rs rec_sched)
  (online_reg program) (reg_db database)
  (terminal_a terminal)
  (registrars_office loc) (anyplace loc)
)
:i '(
)
:g '(
(knows_h (basil reg_help_info))
)
)

```

```

;-----
(def_req "inaccurate_student_info"
:d "A student's information should be accurate"
:r 'bad
:o '(
  (abigail person)(basil person)
  (registrar_a registrar)(advisor_a advisor)
  (basils_trans_info transcript_info)
  (basils_sched_info sched_info)
  (basils_id id)(basils_transcript transcript)
  (basils_pac pac)
  (win93 term)(CIS_BS degree)
  (the_school_info school_info)(reg_help_info help_info)
  (basils_st_info_1 student_info)
  (basils_st_info_2 student_info)
  (basils_rs rec_sched)
  (online_reg program)(reg_db database)
  (terminal_a terminal)
  (registrars_office loc)(anyplace loc)
)
:i '(
(diff_info (basils_st_info_1 basils_st_info_2))
(diff_info (basils_st_info_2 basils_st_info_1))
(has_student_info (basil basils_st_info_1))
(db_has_si_in (reg_db basils_st_info_1))
(not_db_has_si_in (reg_db basils_st_info_2))
)
:g '(
(has_student_info (basil basils_st_info_2))
(not_db_has_si_in (reg_db basils_st_info_2))
)
)

```



Object Definitions

```
(def_otyp "faculty"
  :sb '(advisor))

(def_otyp "student"
  :sb '(undergrad grad))

(def_otyp "person"
  :sb '(student faculty registrar))

;(def_otyp "person"
;:sb '(student faculty ))

(def_otyp "document"
  :sb '(id transcript schedule_listing catalog) )

(def_otyp "terminal"
  :sb '(window_terminal line_terminal) )

(setf *info_otyp*
  (def_otyp "info"
    :sb '(transcript_info sched_info pac rec_sched
          school_info help_info student_info))
  )

(def_otyp "thing"
  :sb '(person info program database terminal term degree
        database terminal class seat loc document) )
```

The definitions on this page are domain information used to derive privacy requirements and used with the motivated agent heuristic.

```
(setf *has_info_objs* '(person transcript_info degree term rec_sched
sched_info pac student_info))
```

```
(setf *has_ptyp*
  (def_ptyp "has_info" :o *has_info_objs*
    :sb '(("has_transcript_info" (person transcript_info))
          ("has_rec_sched" (person degree term rec_sched))
          ("has_schedule_info" (person term sched_info))
          ("has_student_info" (person student_info))
          ("has_pac" (person pac)) ))
  )
```

```
(setf *knows_objs* '(person transcript_info rec_sched sched_info pac
student_info school_info help_info))
```

```
(setf *knows_ptyp*
  (def_ptyp "knows_info" :o *knows_objs*
    :sb '(("knows_t" (person transcript_info))
          ("knows_r" (person rec_sched))
          ("knows_s" (person sched_info))
          ("knows_p" (person pac))
          ("knows_c" (person school_info))
          ("knows_h" (person help_info))
          ))
  )
```

## APPENDIX C

### EXAMPLES

The first section of this appendix gives examples from transcripts of GIRAFFE sessions. The second section gives examples of capabilities that can be added to a specification. The third section shows examples of requirements as they are output by GIRAFFE.

#### Examples from Traces

These examples from traces of GIRAFFE's execution show three kinds of transformations: adding constraints to an achievement requirement, adding constraints to a safety requirement, and making a disjunction in the initial state qualifications.

#### Detailed Example of Transformations

In this example I show how the <find out schedule> requirement is transformed as the client changes the specification of the artifact. Figure 45 gives an overview of the steps described in this example.

(1S) The specification of the artifact is empty, and the only capabilities available are those present in the artifact of the environment. Some examples of capabilities present in the environment are the capabilities for people to go from place to place, read documents, tell information, and so on.

(1R) The initial version of <find out schedule> is the impractical, ideal version with no qualifications on the initial state or the path:

```
-----
REQUIREMENT <find_out_schedule>      achievement NOT Satisfied
A student should be able to find out his/her schedule information.
```

```
Achieve a state where:
(has_schedule_info (basil win93 basils_sched_info))
(knows_s (basil basils_sched_info))
```

```
From a state where:
```

```
Constraints on actions:
-----
```

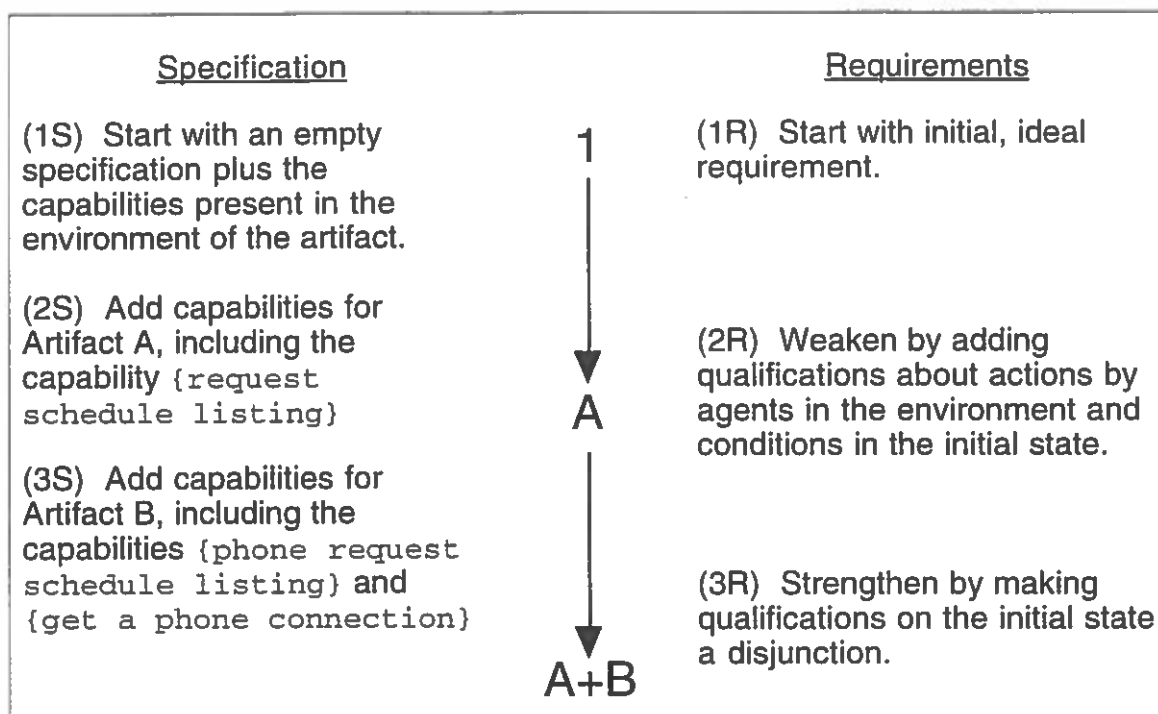


FIGURE 45. A summary of some specification changes and requirements transformations.

(2S) The client adds the capabilities for Artifact A (the in-person registration system) to the specification.

(2R) When the client adds the capabilities for Artifact A, GIRAFFE transforms the requirements in two ways: first by adding constraints on actions performed by agents in the environment, and second by qualifying the initial state.

The constraints on environmental agent actions described in this example are similar to the first example in Figure 35, where the qualification is an assumption that some agent in the environment will perform some action. The constraints are suggested by the `X_strengthen_ea_pc` transformation rule. (See Chapter IV for a description of the transformation rules.) The applicability conditions of the rule are stated in terms of scenarios, so in order for GIRAFFE to decide whether the rule applies it must find out what scenarios are possible for the requirement's transition. In this case, no scenarios are possible without changing the constraint and GIRAFFE applies the rule.

The rule finds two transformations (instantiations of the rule) that represent different ways of changing the constraints on agents in the environment. In the first transformation, an environmental agent (presumably the student) is responsible for going to the registrar's office, requesting a schedule listing and reading the information. In the second transformation, the student is only responsible for reading the schedule information. The following output shows how GIRAFFE describes the two transformations:

## TRANSFORMATIONS FOR &lt;find\_out\_schedule&gt;

```
-----
<xf18> state assumptions for 3 env. agent actions in <get_schedule_i..
R# -1
Weaken the requirement by assuming that an agent in the environment will
perform the following actions:
{go to registrars office}           {request schedule listing}
{read schedule doc}
```

## Rating Summary (R#): -1

|                       |   |    |                        |   |    |
|-----------------------|---|----|------------------------|---|----|
| worst_max_agents      | t | -1 | best_min_agents        | t | 1  |
| max_agents            | 2 | 0  | min_agents             | 2 | 0  |
| number_of_etyps       | 3 | -3 | number of scenarios    | 2 | 1  |
| satisfied by spec?    | t | 1  | worst maximum duration | t | -1 |
| best minimum duration | t | 1  | maximum duration       | 4 | 0  |
| minimum duration      | 4 | 0  |                        |   |    |

```
-----
<xf17> state assumptions for 1 env. agent action in <get_schedule_in..
R# 0
```

```
Weaken the requirement by assuming that an agent in the environment will
perform the following actions:
{read schedule doc}
```

## Rating Summary (R#): 0

|                       |   |    |                        |   |    |
|-----------------------|---|----|------------------------|---|----|
| worst_max_agents      | t | -1 | best_min_agents        | t | 1  |
| max_agents            | 2 | 0  | min_agents             | 2 | 0  |
| number_of_etyps       | 1 | -1 | number of scenarios    | 1 | 0  |
| satisfied by spec?    | t | 1  | worst maximum duration | t | -1 |
| best minimum duration | t | 1  | maximum duration       | 4 | 0  |
| minimum duration      | 4 | 0  |                        |   |    |

In its descriptions of transformations, GIRAFFE lists the rating functions that are applicable. For each rating function, the program lists the value for the rating function and the numeric rating assigned to that value. For example, for the transformation <xf18> described above, the "number of scenarios" rating functions has a value 9, meaning there are nine scenarios possible, and the numeric rating 1, meaning 1 is added to the total rating of that transformation. The numeric rating is positive because the requirement is an achievement rule, and the more scenarios that are possible, the better. (See Chapter IV for a description rating functions in general and the "number of scenarios" rating function in particular.) Some rating functions, such as "worst maximum agents" (abbreviated as `worst_max_agents`) do not produce numeric values. For those functions, other kinds of values are listed, usually a LISP boolean (t for true and nil for false).

The total rating for each transformation is listed as the "Rating Summary", sometimes abbreviated as "R#". GIRAFFE gives the second transformation a higher rating summary, due to the fact that it subtracts one point from the rating for each action (etyp) required by the environmental agent, but the client gives more weight to the number of scenarios and chooses the first transformation as shown below. Listing each rating function with its value and numeric rating gives the client more information in making a choice, while the rating summary gives GIRAFFE a means of making a choice in the absence of input from the client.

```

SELECT THE TRANSFORMATION(S) YOU WANT TO APPLY TO THIS REQUIREMENT:
1 <xf18> state assumptions for 3 env. agent actions in
<get_schedule_i.. R# -1
2 <xf17> state assumptions for 1 env. agent action in
<get_schedule_in.. R# 0
Enter a list of numbers (or c to cancel): (1)

```

Do you want to weaken the requirement <find\_out\_schedule> by assuming that agents in the environment of the artifact will carry out the following actions:

```

{go to registrars office}          (request schedule listing)
{read schedule doc}

```

Do you want to change the requirement? (y/n) y

Changing the path constraint.

All of the scenarios showing how the transition for <find out schedule> can occur require additional conditions in the initial state. GIRAFFE therefore suggests qualifying the initial state by adding five conditions. This transformation is similar to the first example shown in Figure 34. The transformation is suggested by the X\_add\_inits transformation rule:

```

-----
<xf38> add 5 conditions to the initial state of <find_out_schedule>
R# 1

```

Add initial conditions:

```

+ (has_schedule_info (basil win93 basils_sched_info))
+ (db_has_s_in (oa_db basils_sched_info))
+ (person_has_access_to (registrar_a oa_db))
+ (works_at_registrars (registrar_a))
+ (not_sent_schedule (basil))

```

Rating Summary (R#): 1

|                     |   |    |                    |   |   |
|---------------------|---|----|--------------------|---|---|
| worst_max_agents    | t | -1 | best_min_agents    | t | 1 |
| max_agents          | 2 | 0  | min_agents         | 2 | 0 |
| number of scenarios | 1 | 0  | satisfied.by spec? | t | 1 |
| worst max duration  | t | -1 | best min duration  | t | 1 |
| maximum duration    | 4 | 0  | minimum duration   | 4 | 0 |

```

-----
SELECT THE TRANSFORMATION(S) YOU WANT TO APPLY TO THIS REQUIREMENT:
1 <xf38> add 5 conditions to the initial state of <find_out_schedule>
R# 1
Enter a list of numbers (or c to cancel): (1)

```

Adding the following conditions enables a scenario for requirement <find\_out\_schedule>:

```

{has_schedule_info}(basil win93 basils_sched_info)
{db_has_s_in}(oa_db basils_sched_info)
{person_has_access_to}(registrar_a oa_db)
{works_at_registrars}(registrar_a)
{not_sent_schedule}(basil)

```

These conditions are not part of the current version of the requirement <find\_out\_schedule>.

Do you want to add these initial conditions to the requirement? y/n y

Adding the following initial conditions to requirement  
 <find\_out\_schedule>:  
 (has\_schedule\_info (basil win93 basils\_sched\_info))  
 (db\_has\_s\_in (oa\_db basils\_sched\_info))  
 (person\_has\_access\_to (registrar\_a oa\_db))  
 (works\_at\_registrars (registrar\_a))  
 (not\_sent\_schedule (basil))

---

The client accepts the transformation, and GIRAFFE applies it to give the following version of <find out schedule> for Artifact A:

---

REQUIREMENT <find\_out\_schedule>            achievement    SATISFIED  
 A student should be able to find out his/her schedule information.

Achieve a state where:  
 (has\_schedule\_info (basil win93 basils\_sched\_info))  
 (knows\_s (basil basils\_sched\_info))

From a state where:  
 (has\_schedule\_info (basil win93 basils\_sched\_info))  
 (db\_has\_s\_in (reg\_db basils\_sched\_info))  
 (person\_has\_access\_to (registrar\_a reg\_db))  
 (works\_at\_registrars (registrar\_a))  
 (not\_sent\_schedule (basil))

Constraints on actions:  
 > All scenarios require basil to perform the action {read schedule doc}.  
 > Ignore scenarios with any actions performed by agents in the  
 environment except for the following actions:  
     {go to registrars office}                      {request schedule listing}  
     {read schedule doc}

> Ignore scenarios with no actions.

---

(3S) The client adds capabilities for Artifact B (phone registration) to the specification.

(3R) Now there is an additional way for a student to find out her schedule: she can call the registration program. GIRAFFE therefore suggests changing the qualification on the initial state to a disjunction. This change is a way of strengthening the requirement by weakening the qualification. The transformation is suggested by the X\_add\_disj transformation rule.

```
-----
<xf37> add 4 conditions to a disjunction in the initial state of <ge..
R# 1
```

```
Add conditions to a new disjunction in the initial state:
```

```
+ (has_access_to (online_reg reg_db))
+ (has_phone_access (basil))
+ (can_do_phone_io (online_reg))
+ (connection_available (online_reg))
```

```
Rating Summary (R#): 1
```

|                       |   |    |                        |   |    |
|-----------------------|---|----|------------------------|---|----|
| worst max agents      | t | -1 | best min agents        | t | 1  |
| maximum agents        | 2 | 0  | minimum agents         | 2 | 0  |
| number of scenarios   | 2 | 1  | worst maximum duration | t | -1 |
| best minimum duration | t | 1  | maximum duration       | 3 | 0  |
| minimum duration      | 2 | 0  |                        |   |    |

```
-----
```

The ratings indicate whether this transformation leads to a stronger, satisfied requirement. In this case, the most important rating is the number of scenarios, because it indicates that the requirement will still be satisfied even though the qualification is weaker. The following more detailed description of the transformation indicates to the client that the requirement will be stronger after the transformation occurs:

```
Application of X_add_disj to find_out_schedule
```

```
Adding a disjunction makes the requirement <find_out_schedule> stronger
but still satisfied:
```

```
[
  (not_sent_schedule (basil))
  (works_at_registrars (registrar_a))
  (person_has_access_to (registrar_a reg_db))
]
[
  (has_access_to (online_reg reg_db))
  (has_phone_access (basil))
  (can_do_phone_io (online_reg))
  (connection_available (online_reg))
]
(has_schedule_info (basil win93 basils_sched_info))
(db_has_s_in (reg_db basils_sched_info))
Do you want to change the requirement <find_out_schedule> to include the
disjunction? y
```

The two sets of conditions in brackets are two alternative sets of conditions. The two conditions after the brackets are common conditions required for every known scenario. The disjunction shown here is similar to the one shown in the second example in Figure 34, except that in this case there are more conditions in each disjunct.



The final version of the requirement (for this example) shows the weakened qualification that includes a disjunction:

```

-----
REQUIREMENT <find_out_schedule>      achievement  SATISFIED
A student should be able to find out his/her schedule information.

Achieve a state where:
(has_schedule_info (basil win93 basils_sched_info))
(knows_s (basil basils_sched_info))

From a state where:
  [
    (not_sent_schedule (basil))
    (works_at_registrars (registrar_a))
    (person_has_access_to (registrar_a reg_db))
  ]
  [
    (has_access_to (online_reg reg_db))
    (has_phone_access (basil))
    (can_do_phone_io (online_reg))
    (connection_available (online_reg))
  ]
(has_schedule_info (basil win93 basils_sched_info))
(db_has_s_in (reg_db basils_sched_info))

Constraints on actions:
> Ignore scenarios with any actions performed by agents in the
environment except for the following actions:
  {go to registrars office}           {request schedule listing}
  {read schedule doc}                {phone request schedule listing}
  {get a phone connection}
> Ignore scenarios with no actions.
-----

```

### Constraining Actions for a Safety Requirement

These transformations suggest qualifying the <transcript privacy> requirement by adding constraints on various actions. Constraints on actions performed by "motivated agents" are given higher ratings than constraints on actions performed by arbitrary agents. The heuristic of rating actions by motivated agents higher than those performed by other agents is based on the assumption that students will be more likely to constrain their actions to protect their privacy than intruders will be to constrain their actions (see Chapter III). Because of the motivated-agent heuristic, constraints on {steal an ID} are rated less than constraints on {give an ID}.

Another factor in the ratings for these transformations is the number of other requirements for which the constraint is suggested. The constraint on "give an ID" is important for other privacy requirements as well, and so is given a higher rating. The constraint on "go to the registrar's office" is also given a higher rating, though in fact this constraint is not added to the artifacts I studied. A global rating function, *G\_etyp\_xfs*, increases ratings for constraints on actions that are the same. The values for this function are listed as "constrained in others".

Note that it is possible to have an assumption in one requirement that an action will not occur and an assumption in another requirement that the same action will occur.

GIRAFFE uses global rating functions to describe such conflicts, as discussed in Chapter IV. An example of such a rating function is `G_not_etyp_xfs`, described in the listing for `<xf395>` below as "conflicts w/ sugg evt". This rating means that the constraint on the action {go to registrars office} in this transformation, is the opposite of the constraint on the same action in a transformation for a different requirement. Allowing students access to the registrar's office lets them find out their transcript information, but that access could also be used to violate a privacy requirement.

TRANSFORMATIONS FOR `<transcript_privacy>`

-----  
`<xf395>` add a constraint on {go to registrars office} to `<transcript_privacy>`.  
R# 2  
Add a path constraint:  
+ ACTION {go to registrars office} does not occur.

Rating Summary (R#): 2

|                       |   |    |                       |   |   |
|-----------------------|---|----|-----------------------|---|---|
| conflicts w/ sugg evt | t | -1 | constrained in others | 3 | 3 |
| maximum agents        | 1 | 0  | minimum agents        | 1 | 0 |
| number of scenarios   | 1 | 0  | maximum duration      | 1 | 0 |
| minimum duration      | 1 | 0  |                       |   |   |

-----

`<xf396>` add a constraint on {read transcript doc} to `<transcript_privacy>`.  
R# -1  
Add a path constraint:  
+ ACTION {read transcript doc} does not occur.

Rating Summary (R#): -1

|                       |   |    |                     |   |   |
|-----------------------|---|----|---------------------|---|---|
| conflicts w/ sugg evt | t | -1 | maximum agents      | 1 | 0 |
| minimum agents        | 1 | 0  | number of scenarios | 1 | 0 |
| maximum duration      | 1 | 0  | minimum duration    | 1 | 0 |

-----

`<xf397>` add a constraint on {give an id} to `<transcript_privacy>`.  
R# 2  
Add a path constraint:  
+ ACTION {give an id} does not occur.

Rating Summary (R#): 2

|                       |   |    |                  |   |   |
|-----------------------|---|----|------------------|---|---|
| constrained in others | 1 | 1  | motivated agent  | t | 2 |
| maximum agents        | 3 | 0  | minimum agents   | 1 | 0 |
| number of scenarios   | 9 | -1 | maximum duration | 5 | 0 |
| minimum duration      | 1 | 0  |                  |   |   |

-----

-----  
 <xf398> add a constraint on {steal an id} to <transcript\_privacy>

R# 0

Add a path constraint:

+ ACTION {steal an id} does not occur.

Rating Summary (R#): 0

|                       |   |   |  |                     |   |    |
|-----------------------|---|---|--|---------------------|---|----|
| constrained in others | 1 | 1 |  | maximum agents      | 3 | 0  |
| minimum agents        | 1 | 0 |  | number of scenarios | 9 | -1 |
| maximum duration      | 5 | 0 |  | minimum duration    | 1 | 0  |

-----

<xf399> add a constraint on {tell transcript info} to <transcript\_pri..

R# 1

Add a path constraint:

+ ACTION {tell transcript info} does not occur.

Rating Summary (R#): 1

|                  |   |   |  |                     |   |    |
|------------------|---|---|--|---------------------|---|----|
| motivated agent  | t | 2 |  | maximum agents      | 3 | 0  |
| minimum agents   | 2 | 0 |  | number of scenarios | 9 | -1 |
| maximum duration | 5 | 0 |  | minimum duration    | 5 | 0  |

-----

<xf400> add a constraint on {give a transcript} to <transcript\_privacy>

R# 1

Add a path constraint:

+ ACTION {give a transcript} does not occur.

Rating Summary (R#): 1

|                  |   |   |  |                     |    |    |
|------------------|---|---|--|---------------------|----|----|
| motivated agent  | t | 2 |  | maximum agents      | 3  | 0  |
| minimum agents   | 1 | 0 |  | number of scenarios | 11 | -1 |
| maximum duration | 5 | 0 |  | minimum duration    | 1  | 0  |

-----

<xf401> add a constraint on {steal a transcript} to <transcript\_priva..

R# -1

Add a path constraint:

+ ACTION {steal a transcript} does not occur.

Rating Summary (R#): -1

|                     |    |    |  |                  |   |   |
|---------------------|----|----|--|------------------|---|---|
| maximum agents      | 3  | 0  |  | minimum agents   | 1 | 0 |
| number of scenarios | 11 | -1 |  | maximum duration | 5 | 0 |
| minimum duration    | 1  | 0  |  |                  |   |   |

-----

<xf402> add a constraint on {request official transcript} to <transcr..

R# -1

Add a path constraint:

+ ACTION {request official transcript} does not occur.

Rating Summary (R#): -1

|                       |   |    |  |                     |   |   |
|-----------------------|---|----|--|---------------------|---|---|
| conflicts w/ sugg evt | t | -1 |  | maximum agents      | 1 | 0 |
| minimum agents        | 1 | 0  |  | number of scenarios | 1 | 0 |
| maximum duration      | 1 | 0  |  | minimum duration    | 1 | 0 |

-----

```

SELECT THE TRANSFORMATION(S) YOU WANT TO APPLY TO THIS REQUIREMENT:
1 <xf395> add a constraint on {go to registrars office} to
<transcript_..      R# 2
2 <xf396> add a constraint on {read transcript doc} to
<transcript_priv..  R# -1
3 <xf397> add a constraint on {give an id} to <transcript_privacy>
R# 2
4 <xf398> add a constraint on {steal an id} to <transcript_privacy>
R# 0
5 <xf399> add a constraint on {tell transcript info} to
<transcript_pri..   R# 1
6 <xf400> add a constraint on {give a transcript} to
<transcript_privacy> R# 1
7 <xf401> add a constraint on {steal a transcript} to
<transcript_priva.. R# -1
8 <xf402> add a constraint on {request official transcript} to
<transcr..          R#-1
Enter a list of numbers (or c to cancel):

```

### Artifact Capability

This section lists examples of capabilities used in the specification of on-line registration systems. Most of the capabilities listed here are used in Artifact B, which provides phone registration. The first capability, {request schedule listing}, is used in Artifact A. These capabilities are in the form used by the GIRAFFE program. For a description of the format of capabilities, see the description given for environmental capabilities given in Appendix B.

```

;----- STUDENT ACTIONS -----
; User actions enabled by system capabilities.

(def_etyp "request schedule listing"
:d "a student requests a schedule of his/her schedule"
:o '(person1 person2 sched_infol term1)
:= '((at_registrars_office (person1))
      (has_schedule_info (person2 term1 sched_infol))) )
:+ '((requested_schedule_listing (person1 sched_infol))
      (^requested_schedule_listing (person1 sched_infol)) )
:at '((category :val env_agent))
)

(def_etyp "get a phone connection"
:d "a person gets a phone connection to a program"
:o '(person1 person2 program1)
:- '((connection_available (program1)))
:= '((can_do_phone_IO (program1))
      (has_phone_access (person1)) )
:+ '((has_phone_connection (person1 person2 program1))
      (_is_+phone_connect) )
:at '((category :val env_agent))
)

```

```

(def_etyp "get a PAC phone connection"
:d "a person gets a PAC phone connection to a program"
:o '(person1 person2 program1 person1 person2 pac1)
:- '((connection_available (program1)))
:= '((can_do_phone_IO (program1))
      (has_pac (person2 pac1))
      (knows_p (person2 pac1))
      (has_phone_access (person1)) )
:+ '((has_phone_connection (person1 person2 program1))
      (_is_+pac_phone_connect) )
:at '((category :val env_agent))
)

(def_etyp "phone request add"
:d "a student requests addition of a class to schedule"
:o '(person1 person2 program1 class1 term1)
:= '((has_phone_connection (person1 person2 program1))
      (wants (person2 class1 term1)) )
:+ '((ph_requested_add (person2 class1 term1))
      (^requested_add (person2 class1 term1)) )
:at '((category :val env_agent))
)

(def_etyp "phone request wrong add"
:d "a student requests addition of the wrong class"
:o '(person1 person2 program1 class1 term1)
:= '((has_phone_connection (person1 person2 program1))
      (not_wants (person2 class1 term1)) )
:+ '((ph_requested_add (person2 class1 term1)))
:at '((category :val env_agent))
)

(def_etyp "phone request drop"
:d "a student requests drop for a class"
:o '(person1 person2 program1 class1 term1)
:= '((has_phone_connection (person1 person2 program1))
      (not_wants (person2 class1 term1)) )
:+ '((ph_requested_drop (person2 class1 term1))
      (^requested_drop (person2 class1 term1)) )
:at '((category :val env_agent))
)

(def_etyp "phone request transcript info"
:d "a student requests an transcript information"
:o '(person1 person2 program1 transcript_info1)
:= '((has_phone_connection (person1 person2 program1))
      (has_transcript_info (person2 transcript_info1)) )
:+ '((ph_requested_transcript_info (person2 transcript_info1))
      (^requested_transcript (person2 transcript_info1)) )
:at '((category :val env_agent))
)

```

```

(def_etyp "phone request schedule listing"
:d "a student requests a schedule of his/her schedule"
:o '(person1 person2 program1 sched_info1 term1)
:= '((has_phone_connection (person1 person2 program1))
      (has_schedule_info (person2 term1 sched_info1)) )
:+ '((ph_requested_schedule_listing (person2 sched_info1))
      (^requested_schedule_listing (person2 sched_info1)) )
:at '((category :val env_agent))
)

;----- DEPT/REG ACTIONS -----

(def_etyp "phone give transcript info"
:d "program tells transcript info to a student over the phone"
:o '(program1 person1 person2 transcript_info1 databasel transcript1)
:- '((ph_requested_transcript_info (person2 transcript_info1))
:= '((can_do_phone_IO (program1))
      (has_phone_connection (person1 person2 program1))
      (has_access_to (program1 databasel))
      (db_has_t_in (databasel transcript_info1)) )
:+ '((knows_t (person2 transcript_info1))
      (_is+_ph_transcript ()) )
:at '((category :val art_agent))
)

(def_etyp "phone give schedule info"
:d "program tells a schedule listing to a student over the phone"
:o '(program1 person1 person2 sched_info1 term1 databasel
schedule_listing1)
:- '((ph_requested_schedule_listing (person2 sched_info1))
:= '((can_do_phone_IO (program1))
      (has_phone_connection (person1 person2 program1))
      (has_access_to (program1 databasel))
      (db_has_s_in (databasel sched_info1)) )
:+ '((knows_s (person2 sched_info1))
      (_is+_ph_schedule ()) )
:at '((category :val art_agent))
)

(def_etyp "phone add class"
:d "program adds class for a student"
:o '(program1 person1 person2 term1 class1 seat1 sched_info1 databasel)
:- '((ph_requested_add (person2 class1 term1))
      (seat_available (seat1 class1 term1)) )
:= '((can_do_phone_IO (program1))
      (has_phone_connection (person1 person2 program1))
      (has_access_to (program1 databasel))
      (db_has_s_in (databasel sched_info1))
      (has_schedule_info (person2 term1 sched_info1)) )
:+ '((class_in_sched (class1 sched_info1))
:at '((category :val art_agent))
)

```

```

(def_etyp "phone drop class"
:d "program drops class for a student"
:o '(program1 person1 person2 term1 class1 seat1 sched_infol database1)
:- '((ph_requested_drop (person2 class1 term1))
      (class_in_sched (class1 sched_infol)) )
:= '((can_do_phone_IO (program1))
      (has_phone_connection (person1 person2 program1))
      (has_schedule_info (person2 term1 sched_infol))
      (db_has_s_in (database1 sched_infol))
      (has_access_to (program1 database1)) )
:+ '((seat_available (seat1 class1 term1))
      (not_class_in_sched (class1 sched_infol)) )
:at '((category :val art_agent))
)

(def_etyp "PAC phone give transcript info"
:d "program tells transcript info to a student over the phone"
:o '(program1 person1 person2 transcript_infol database1 transcript1)
:- '((ph_requested_transcript_info (person2 transcript_infol))
:= '((can_do_phone_IO (program1))
      (has_phone_connection (person1 person2 program1))
      (has_transcript_info (person2 transcript_infol))
      (has_access_to (program1 database1))
      (db_has_t_in (database1 transcript_infol)) )
:+ '((knows_t (person2 transcript_infol))
      (_is+_PAC_ph_transcript ())) )
:at '((category :val art_agent))
)

(def_etyp "PAC phone add class"
:d "program adds class for a student"
:o '(program1 person1 person2 term1 class1 seat1 sched_infol database1)
:- '((ph_requested_add (person2 class1 term1))
      (seat_available (seat1 class1 term1)) )
:= '((can_do_phone_IO (program1))
      (has_phone_connection (person1 person2 program1))
      (has_schedule_info (person2 term1 sched_infol))
      (has_access_to (program1 database1))
      (db_has_s_in (database1 sched_infol))
      (has_schedule_info (person2 term1 sched_infol)) )
:+ '((class_in_sched (class1 sched_infol))
      (_is+_PAC_ph_add_class ())) )
:at '((category :val art_agent))
)

;----- CHANGE PACs -----
(def_etyp "change own pac"
:d "a person changes his/her own PAC when someone else knows it"
:o '(person1 person2 pac1)
:- '((knows_p (person2 pac1))
:= '((knows_p (person1 pac1))
      (has_PAC (person1 pac1))
      (diff (person1 person2)) )
:+ '((not_knows_p (person2 pac1))
:at '((category :val env_agent))
)

```

```

(def_etyp "ID tell PAC"
:d "registrar tells PAC to a student"
:o '(person1 person2 id1 pac1)
:- '((asked_PAC (person2)))
:= '((works_at_registrars (person1))
      (has_pac (person2 pac1))
      (has_id (person2 id1))
      (is_ID_for (id1 person2))
      (at_registrars_office (person2)) )
:+ '((knows_p (person2 pac1)))
:at '((category :val art_agent))
)

(def_etyp "assign hard PAC"
:d "registrar assigns a hard-to-guess PAC to a student"
:o '(person1 pac1)
:- '((is_uknown_diff (pac1)))
:= '((works_at_registrars (person1)))
:+ '((is_hard_pac (pac1)))
:at '((category :val art_agent))
)

(def_etyp "assign easy PAC"
:d "registrar assigns an easy-to-guess PAC to a student"
:o '(person1 pac1)
:- '((is_uknown_diff (pac1)))
:= '((works_at_registrars (person1)))
:+ '((is_easy_pac (pac1)))
:at '((category :val art_agent))
)

```

### Requirements for Artifact B

This section gives examples of requirements derived by GIRAFFE for Artifact B, a registration system that allows registration transactions by phone. The requirements are in one of two formats that GIRAFFE uses to print out requirements. Each requirement listing shows the final state, initial state, and path constraints.

Achievement requirements are listed as SATISFIED if there is at least one scenario showing how the transition for the requirement can occur. Safety requirements are listed as SATISFIED if there are no scenarios for the transition.

For purposes of determining satisfaction of requirements, GIRAFFE ignores scenarios that don't satisfy path constraints, so path constraints describing scenarios that GIRAFFE ignores are describing violations of path constraints. So for example, if GIRAFFE ignores scenarios with the action {tell PAC}, there is a constraint of the form [does\_not\_have\_action tell\_PAC].

Note that since these requirements are printed by GIRAFFE, they are not in the same format as the initial requirements listed in Appendix B.



The format for the sample listing of requirements below does not include object types. GIRAFFE's other option for listing requirements produces a listing like the following for each requirement:

```
(abigail person)                (basil person)
(registrar_a registrar)         (advisor_a advisor)
(cis121 class)                  (121seat seat)
(basils_trans_info transcript_info) (basils_sched_info sched_info)
(basils_id id)                  (basils_transcript transcript)
(basils_st_info student_info)   (basils_pac pac)
(win93 term)                    (cis_bs degree)
(basils_rs rec_sched)           (online_reg program)
(reg_db database)               (terminal_a terminal)
(registrars_office loc)         (anyplace loc)
```

Most of the requirements in this section have a similar list of objects.

```
-----
REQUIREMENT <add_class>                achievement SATISFIED
A student should be able to add a class.
```

Achieve a state where:

```
(has_schedule_info (basil win93 basils_sched_info)) (class_in_sched
(cis121 basils_sched_info))
```

From a state where:

```
(wants (basil cis121 win93))
(has_schedule_info (basil win93 basils_sched_info))
(db_has_s_in (reg_db basils_sched_info))
(person_has_access_to (registrar_a reg_db))
(works_at_registrars (registrar_a))
(at (basil anyplace))
(seat_available (121seat cis121 win93))
(has_schedule_info (basil win93 basils_sched_info))
(db_has_s_in (reg_db basils_sched_info))
(has_access_to (online_reg reg_db))
(has_phone_access (basil))
(can_do_phone_io (online_reg))
(connection_available (online_reg))
(seat_available (121seat cis121 win93))
```

Constraints on actions:

> Ignore scenarios with any actions performed by agents in the environment except for the following actions:

```
{go to registrars office}          {request add}
{get a phone connection}          {phone request add}
```

> Ignore scenarios with no actions.

```
-----
```

```
-----
REQUIREMENT <find_out_schedule>      achievement  SATISFIED
A student should be able to find out his/her schedule information.
```

Achieve a state where:

```
(has_schedule_info (basil win93 basils_sched_info)) (knows_s (basil
basils_sched_info))
```

From a state where:

```
(has_schedule_info (basil win93 basils_sched_info))
(db_has_s_in (reg_db basils_sched_info))
(person_has_access_to (registrar_a reg_db))
(works_at_registrars (registrar_a))
(not_sent_schedule (basil))
(db_has_s_in (reg_db basils_sched_info))
(has_access_to (online_reg reg_db))
(has_schedule_info (basil win93 basils_sched_info))
(has_phone_access (basil))
(can_do_phone_io (online_reg))
(connection_available (online_reg))
```

Constraints on actions:

> Ignore scenarios with any actions performed by agents in the environment except for the following actions:

```
{read schedule doc}           {get a phone connection}
{phone request schedule listing}
```

> Ignore scenarios with no actions.

```
-----
REQUIREMENT <unwanted_class>          safety    SATISFIED
A student should not have an unwanted class.
```

Do NOT allow achievement of a state where:

```
(has_schedule_info (basil win93 basils_sched_info)) (class_in_sched
(cis121 basils_sched_info))
```

From a state where:

```
(not_wants (basil cis121 win93))
```

Constraints on actions:

> Ignore scenarios in which ALL of the following ACTIONS occur:

```
{get a phone connection}           {phone request wrong add}
```

> Ignore scenarios in which the ACTION {go to registrars office} occurs.

In the <unwanted class> requirement above, the path constraint is a complex one (see Chapter II):

```
[does_not_have_action {get a phone connection}] ^
[does_not_have_action {phone request wrong add}].
```

The path constraint is only violated when both of the actions occur.

-----  
REQUIREMENT <get\_advice> achievement SATISFIED  
A student should be able to find out his/her recommended schedule.

Achieve a state where:  
(has\_rec\_sched (basil cis\_bs win93 basils\_rs)) (knows\_r (basil  
basils\_rs))

From a state where:  
(person\_has\_access\_to (advisor\_a reg\_db))  
(db\_has\_t\_in (reg\_db basils\_trans\_info))  
(has\_rec\_sched (basil cis\_bs win93 basils\_rs))  
(has\_transcript\_info (basil basils\_trans\_info))  
(can\_advise (advisor\_a))  
(needs\_advisor (basil))  
(has\_time\_available (advisor\_a))  
(at (basil anyplace))

Constraints on actions:  
> Ignore scenarios with any actions performed by agents in the  
environment except for the following actions:  
    {get an appointment}                      {go to advisors office}  
> Ignore scenarios with no actions.

-----  
REQUIREMENT <get\_transcript> achievement SATISFIED  
A student should be able to get a copy of his/her transcript

Achieve a state where:  
(has\_transcript (basil basils\_transcript)) (doc\_has\_t\_info  
(basils\_transcript basils\_trans\_info))

From a state where:  
(db\_has\_t\_in (reg\_db basils\_trans\_info))  
(person\_has\_access\_to (registrar\_a reg\_db))  
(is\_id\_for (basils\_id basil))  
(has\_id (basil basils\_id))  
(works\_at\_registrars (registrar\_a))  
(has\_transcript\_info (basil basils\_trans\_info))  
(at (basil anyplace))

Constraints on actions:  
> Ignore scenarios with any actions performed by agents in the  
environment except for the following actions:  
    {go to registrars office}                      {request official transcript}  
> Ignore scenarios with no actions.

-----  
 REQUIREMENT <get\_transcript\_info> achievement SATISFIED  
 A student should be able to find out his/her transcript information.

Achieve a state where:

(has\_transcript\_info (basil basils\_trans\_info)) (knows\_t (basil  
 basils\_trans\_info))

From a state where:

(db\_has\_t\_in (reg\_db basils\_trans\_info))  
 (has\_access\_to (online\_reg reg\_db))  
 (has\_transcript\_info (basil basils\_trans\_info))  
 (has\_phone\_access (basil))  
 (can\_do\_phone\_io (online\_reg))  
 (connection\_available (online\_reg))  
 (diff (abigail basil))  
 (db\_has\_t\_in (reg\_db basils\_trans\_info))  
 (has\_access\_to (online\_reg reg\_db))  
 (has\_transcript\_info (basil basils\_trans\_info))  
 (has\_phone\_access (abigail))  
 (can\_do\_phone\_io (online\_reg))  
 (connection\_available (online\_reg))

Constraints on actions:

> Ignore scenarios with any actions performed by agents in the  
 environment except for the following actions:  
 {get a phone connection} {phone request transcript info}  
 {tell transcript info}  
 > Ignore scenarios with no actions.

-----  
 REQUIREMENT <get\_school\_info> achievement SATISFIED  
 A student should be able to find out information about his/her school

Achieve a state where:

(knows\_c (basil the\_school\_info))

From a state where:

(at (basil anyplace))  
 (has\_money (basil))

Constraints on actions:

> Ignore scenarios with any actions performed by agents in the  
 environment except for the following actions:  
 {go to registrars office} {buy a catalog}  
 {read catalog doc}  
 > Ignore scenarios with no actions.

```
-----
REQUIREMENT <get_help_info>          achievement SATISFIED
A student should be able to find out help information
```

```
Achieve a state where:
(knows_h (basil reg_help_info))
```

```
From a state where:
(at (basil anyplace))
(has_money (basil))
```

```
Constraints on actions:
> Ignore scenarios with any actions performed by agents in the
environment except for the following actions:
  (go to registrars office)      (buy a catalog)
  (read catalog doc)
> Ignore scenarios with no actions.
```

```
-----
REQUIREMENT <inaccurate_student_info> safety NOT satisfied
A student's information should be accurate
```

```
Do NOT allow achievement of a state where:
(has_student_info (basil basils_st_info_2)) (not_db_has_si_in (reg_db
basils_st_info_2))
```

```
From a state where:
(diff_info (basils_st_info_1 basils_st_info_2))
(diff_info (basils_st_info_2 basils_st_info_1))
(has_student_info (basil basils_st_info_1))
(db_has_si_in (reg_db basils_st_info_1))
(not_db_has_si_in (reg_db basils_st_info_2))
```

```
Constraints on actions:
-----
```

The safety requirement <inaccurate\_student\_info> is not satisfied; there are scenarios showing how safety violations can occur. However, there is a repair requirement derived from this requirement, called <repair\_not\_db\_has\_si\_in>, that is satisfied.

The following requirements are derived requirements. The names and descriptions of these requirements are generated by the program and some are therefore difficult to read or less informative than the names and descriptions for requirements in the initial set.

```
-----
REQUIREMENT <sched_info_privacy>      safety   SATISFIED
An intruder should not be able to find out someone else's sched_info
```

```
Do NOT allow achievement of a state where:
(knows_s (intruder basils_sched_info)) (has_schedule_info (basil win93
basils_sched_info))
```

```
From a state where:
(diff (basil intruder))
(diff (intruder basil))
```

```
Constraints on actions:
-----
```

```
-----
REQUIREMENT <repair_class_in_sched>  repair   SATISFIED
Repair by achieving the condition: not_class_in_sched from:
class_in_sched
```

```
Achieve a state where:
(not_class_in_sched (cis121 basils_sched_info))
```

```
From a state where:
(has_schedule_info (basil win93 basils_sched_info))
(class_in_sched (cis121 basils_sched_info))
(not_wants (basil cis121 win93))
(person_has_access_to (registrar_a reg_db))
(db_has_s_in (reg_db basils_sched_info))
(works_at_registrars (registrar_a))
(at (basil anyplace))
```

```
Constraints on actions:
> Ignore scenarios with any actions performed by agents in the
environment except for the following actions:
  {go to registrars office}      {request drop}
> Ignore scenarios with no actions.
-----
```

```
-----
REQUIREMENT <rec_sched_privacy>      safety   SATISFIED
An intruder should not be able to find out someone else's rec_sched
```

```
Do NOT allow achievement of a state where:
(knows_r (intruder basils_rs))      (has_rec_sched (basil cis_bs win93
basils_rs))
```

```
From a state where:
(diff (basil intruder))
(diff (intruder basil))
```

```
Constraints on actions:
-----
```

-----  
 REQUIREMENT <transcript\_info\_privacy> safety SATISFIED  
 An intruder should not be able to find out someone else's  
 transcript\_info

Do NOT allow achievement of a state where:  
 (knows\_t (intruder basils\_trans\_info)) (has\_transcript\_info (basil  
 basils\_trans\_info))

From a state where:  
 (diff (basil intruder))  
 (diff (intruder basil))

Constraints on actions:

- > Ignore scenarios in which the ACTION {give a transcript} occurs.
  - > Ignore scenarios in which the ACTION {tell transcript info} occurs.
  - > Ignore scenarios in which the ACTION {give an id} occurs.
  - > Ignore scenarios in which the ACTION {steal a transcript} occurs.
  - > Ignore scenarios in which the ACTION {steal an id} occurs.
  - > Ignore scenarios in which the ACTION {guess pac} occurs.
- 

In the derived requirement <transcript\_info\_privacy>, some of the constraints are performed by motivated agents (as indicated by rating functions) and so can be considered assignment of responsibility. Those actions are {give a transcript}, {tell transcript info} and {give an id}. The other actions are performed by other agents (an intruder) and are not under the control of a motivated agent. For descriptions of the transformations that produce these constraints, see the second example ("Constraining Actions for a Safety Requirement") of the first section of this appendix.

-----  
 REQUIREMENT <repair\_not\_db\_has\_si\_in> repair SATISFIED  
 Repair by achieving the condition: db\_has\_si\_in from: not\_db\_has\_si\_in

Achieve a state where:  
 (db\_has\_si\_in (reg\_db basils\_st\_info\_2))

From a state where:  
 (has\_student\_info (basil basils\_st\_info\_2))  
 (not\_db\_has\_si\_in (reg\_db basils\_st\_info\_2))  
 (person\_has\_access\_to (registrar\_a reg\_db))  
 (works\_at\_registrars (registrar\_a))  
 (has\_student\_info (registrar\_a basils\_st\_info\_2))  
 (at (registrar\_a anyplace))  
 (db\_has\_si\_in (reg\_db o9))

Constraints on actions:

- > Ignore scenarios with any actions performed by agents in the environment except for the following actions:  
     {go to registrars office}                      {request DB update}
  - > Ignore scenarios with no actions.
-

## REFERENCES

- Anderson, J. S. (1993). *Automating requirements engineering using artificial intelligence planning techniques*, (Tech. Rep. No. CIS-TR-93-28). Eugene: University of Oregon, Computer and Information Science Dept.
- Anderson, J. S., & Durney, B. (1993). Using scenarios in deficiency-driven requirements engineering. In *Proceedings of the IEEE International Symposium On Requirements Engineering* (pp. 134-141). Los Alamitos, CA: IEEE Computer Society.
- Anderson, J. S., & Farley, A. M. (1988). Plan abstraction based on operator generalization. In *Proceedings of the Seventh National Conference on Artificial Intelligence* (pp. 100-104). San Mateo, CA: Morgan Kaufmann.
- Anderson, J. S., & Farley, A. M. (1990). *Incremental selection in plan composition*. (Tech. Rep. No. CIS-TR-90-11). Eugene: University of Oregon, Computer and Information Science Dept.
- Anderson, J. S., & Fickas, S. (1989). A proposed perspective shift: Viewing specification design as a planning problem. In *Proceedings of the Fifth International Conference on Software Specification and Design* (pp. 177-184). Los Alamitos, CA: IEEE Computer Society.
- Boose, J. H. (1986). *Expertise transfer for expert system design*. Amsterdam: Elsevier.
- Carbonell, J. G. (1981). Counterplanning: A strategy-based model of adversary planning in real-world situations. *Artificial Intelligence*, 3, 295-329.
- Dardenne, A. (1993). *On the use of scenarios in requirements acquisition* (Tech. Rep. (Tech. Rep. No. CIS-TR-93-17). Eugene: University of Oregon, Computer and Information Science Dept.
- Dardenne, A., Fickas, S., & van Lamsweerde, A. (1991). Goal-directed concept acquisition in requirements elicitation. In *Proceedings of the 6th International Workshop on Software Specification and Design* (pp. 14-21). Los Alamitos, CA: IEEE Computer Society.
- Dardenne, A., Fickas, S., & van Lamsweerde, A. (1993). Goal-directed requirements acquisition. *Science of Computer Programming*, 20, 3-50.
- Feather, M. S. (1987). Language support for the specification and development of composite systems. *ACM Transactions on Programming Languages and Systems*, 9(2) 198-234.
- Fickas, S., & Helm, R. (1991). *Acting responsibly: Reasoning about agents in a multi-agent system* (Tech. Rep. No. CIS-TR-91-02). Eugene: University of Oregon, Computer and Information Science Dept.



- Fickas, S., & Nagarajan, P. (1988, November). Critiquing software specifications: A knowledge-based approach. *IEEE Software*, pp. 37-47.
- Ginsberg, M., & Smith, D. (1988). Reasoning about action II: The qualification problem. *Artificial Intelligence*, 33(3), 311-342.
- Hall, R. J. (1992). Interactive specification acquisition via scenarios: A proposal. Paper presented at the Workshop on Automating Software Design, AAAI-92, San Jose, CA.
- Hall, R. J. (1993). Validation of rule-based reactive systems by sound scenario generalization. In *Proceedings of the Eighth Knowledge-Based Software Engineering Conference* (pp. 30-39). Los Alamitos, CA: IEEE Computer Society.
- Hammond, K. J. (1989). *Case-based planning: Viewing planning as a memory task*. San Diego, CA: Academic Press.
- Hanks, S. (1990). Practical temporal projection. In *Proceedings of AAAI-90* (pp. 158-163). San Mateo, CA: Morgan Kaufmann.
- Harris, D. R., Benner, K. B., Johnson, W. L., & Feather, M. S. (1992). *Final technical report for the requirements/specification facet for KBSA*. Marina Del Rey, CA: USC Information Sciences Institute.
- Helm, B. R., & Fickas, S. (1992). *Scare tactics: Evaluating problem decompositions using failure scenarios*, (Tech. Rep. No. CIS-TR-92-06). Eugene: University of Oregon, Computer and Information Science Dept.
- Herlihy, M. P., & Wing, J. M. (1991). Specifying graceful degradation. *IEEE Transactions on Parallel and Distributed Systems*, 2(1), 93-104.
- Johnson, W. L., & Feather, M. S. (1991). Using evolution transformations to construct specifications. In Lowry & McCartney (Eds.), *Automating software design* (pp. 65-92). Menlo Park, CA: AAAI Press.
- Kaufman, L. D. (1988). *Scenario selection and implementation techniques for scenario-based rapid prototyping* (Tech. Rep. No. SERC-TR-19-F). Gainesville: University of Florida, Software Engineering Research Center.
- Kaufman, L. D., Thebaut, S. M., & Interrante, M. F. (1989). *System modeling for scenario-based requirements engineering* (Tech. Rep. No. SERC-TR-33-F). Gainesville: University of Florida, Software Engineering Research Center.
- Kramer, G. L., & Petersen, E. D. (Eds.) (1991). *Proceedings of the Conference on The Future of Touch-Tone Telephone Technology: Enhancing Academic Support Services*. Provo, UT: Brigham Young University.
- Lubars, M., Potts, C., & Richter, C. (1993). A review of the state of the practice in requirements modeling. In *Proceedings of the IEEE International Symposium On Requirements Engineering* (pp. 2-14). Los Alamitos, CA: IEEE Computer Society.

- McCrohan, D. (1988). *The Life and Times of Maxwell Smart*. New York: St. Martin's Press.
- Mylopoulos, J., Chung, L., & Nixon, B. (1992). Representing and using non-functional requirements: A process-oriented approach. *IEEE Transactions on Software Engineering*, SE-18(6), 483-497.
- Nixon, B. A. (1993). Dealing with performance requirements during the development of information systems. In *Proceedings of the IEEE International Symposium On Requirements Engineering* (pp. 42-49). Los Alamitos, CA: IEEE Computer Society.
- Peters, E. (1984). *Dead man's ransom*. New York: William Morrow and Company.
- Reubenstein, H. B. (1990). *Automated acquisition of evolving informal descriptions* (Tech. Rep. No. 1205). Cambridge: Massachusetts Institute of Technology, Artificial Intelligence Laboratory.
- Rissland, E. L. (1986). *Dimension-based analysis of hypotheticals from Supreme Court oral argument* (Tech. Rep.) Amherst: University of Massachusetts, COINS.
- Rissland, E. L., & Ashley, K. D. (1986). Hypotheticals as heuristic device. In *Proceedings of AAAI-86* (pp. 289-297). San Mateo, CA: Morgan Kaufmann.
- Robinson, W. N. (1993). Automated negotiated design integration: Formal representations and algorithms for collaborative design (Tech. Rep. No. CIS-TR-93-10). Eugene: University of Oregon, Computer and Information Science Dept.
- Swartout, W., & Balzer, R. (1982). On the inevitable intertwining of specification and implementation. *Communications of the ACM*, 25(7) 438-440.
- Weber, D. G. (1988). *Specifications for fault-tolerance* (Tech. Rep. No. 19-3). Ithaca, NY: Odyssey Research Associates.
- Weber, D. G. (1989). Formal specification of fault-tolerance and its relation to computer security. In *Proceedings of the Fifth International Conference on Software Specification and Design* (pp. 273-277). Los Alamitos, CA: IEEE Computer Society.
- Wilensky, R. (1983). *Planning and understanding*. Reading, MA: Addison-Wesley.