RL-TR-92-248
Final Technical Report
October 1992

ARIES: THE REQUIREMENTS/ SPECIFICATION FACET FOR KBSA

USC Information Sciences Institute

David R. Harris, W. Lewis Johnson, Kevin M. Benner, and Martin S. Feather

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-92-248 has been reviewed and is approved for publication.

APPROVED:

DOUGLAS A. WHITE
Project Engineer

FOR THE COMMANDER:

JOHN A. GRANIERO
Chief Scientist
Command, Control, & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (C3CA) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 1992	Final Mar 89 - Jan 92

4. TITLE AND SUBTITLE
ARIES: THE REQUIREMENTS/SPECIFICATION FACET FOR KBSA

5. FUNDING NUMBERS
C - F30602-89-C-0103
PE - 63728F
PR - 2532
TA - 01
WU - 25

6. AUTHOR(S)
David R. Harris, W. Lewis Johnson, Kevin M. Benner, and Martin S. Feather

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey CA 90292-6695

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Laboratory (C3CA)
525 Brooks Rd
Griffiss AFB NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RL-TR-92-248

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Douglas A. White/C3CA/(315) 330-3564

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report describes a knowledge-based assistant for Acquisition of Requirements and Incremental Evolution of Specifications (ARIES). ARIES is a system developed by the University of Southern California Information Sciences Institute and Lockheed Sanders to address the needs of the Knowledge-Based Software Assistant (KBSA) for the early phases of a system life cycle. ARIES supports the evolutionary development of formal system specifications in a multi-presentational environment in which informal application requirements are captured and elaborated using representations familiar to users and applications engineers. ARIES, through the formalization of activities and creation of a knowledge base, enables extensive machine support for evaluation and reuse of system design artifacts. The concepts explored in this effort will be of interest to anyone interested in: 1) what lies ahead for computer aided software engineering; 2) identifying the key issues for automating the software development process; 3) the value of knowledge-based approaches to formal system development; and 4) the benefits of having a wide spectrum of representations available to the design process.

14. SUBJECT TERMS	15. NUMBER OF PAGES
knowledge-based software engineering, software requirements, software specifications, artificial intelligence, computer aided software engineering, formal specifications, formal methods	162
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Contents

1 Overview and Motivation 1

1.1 Specification development for large complex systems 4

1.2 The state-of-the-practice 5

1.3 The ARIES paradigm—requirements acquisition and specification evolution 10

1.3.1 Assumptions about the process 10

1.3.2 The approach to automation 11

1.3.3 The output of the process 11

1.4 History of the ARIES Project 11

1.4.1 The KBSA Report—short term and long term goals 11

1.4.2 The Requirements Assistant 12

1.4.3 The Specification Assistant 13

1.4.4 Motivation for the ARIES project 13

1.5 Project approach 14

1.5.1 Knowledge-based technology 14

1.5.2 Transformation technology 14

1.6 Use of real world examples 15

1.6.1 Developing a road traffic control specification 16

1.6.2 Developing an air traffic control specification 17

1.6.3 Lessons learned 19
1.7 ARIES facilities 21
1.7.1 Presentation tools 21
1.7.2 Reuse tools 22
1.7.3 Reasoning 23
1.7.4 Evolution 23

2 An Example of Use **24**
2.1 The analyst's view of ARIES 25
2.2 Reviewing requirements 27
2.3 Performing a modification 27
2.4 Reviewing the results of the transformation 31
2.5 Summary 33

3 Representation Issues **35**
3.1 Basic underlying semantics 36
3.2 Advanced semantic features 38
3.2.1 Specialization in ARIES 38
3.2.2 Parameterized concepts 40
3.2.3 Higher-order operators 41
3.2.4 Temporal operators 41
3.3 Scope of knowledge represented 41
3.4 Features not supported 42
3.5 Mapping narrow formalisms onto the underlying representation 43
3.5.1 Mapping via abstracted representations 43
3.5.2 Mapping via transformation 47

3.6 Integrating textual and relational representations 47

4 Presentation **51**

4.1 Construction of presentations 51

4.2 Presentation implementation 55

4.2.1 Implemented presentation styles 55

4.2.2 Implemented presentations 60

4.3 Operational and instructional modes 67

4.4 Related work 68

5 Cooperative Requirements Analysis and Reuse **71**

5.1 Folders and workspaces 72

5.2 Folder structuring 74

5.3 Reuse techniques 76

5.3.1 Representation of multiple models 76

5.3.2 Parameterized specifications 79

5.3.3 Reuse through specialization 79

5.3.4 Reuse of higher-order properties 79

5.4 The impact on automated tools 80

5.5 Examples of Reuse 80

5.5.1 Adjusting a use list 80

5.5.2 Merging conflicting definitions 82

6 Automatic Reasoning for Requirements Engineering **86**

6.1 A framework for reasoning 87

6.1.1 Desirable propagation 88

6.1.2 Tractable computation 90

6.2 Approaches to reasoning in ARIES 90
6.3 Automatic constraint analysis 91
6.3.1 Constraint propagation 91
6.3.2 Incremental static analysis 94
6.4 Static analysis tools 95
6.4.1 Static analysis tools 95
6.5 ARIES Simulation Component 95
6.5.1 Validation questions 97
6.5.2 Influence Analysis 103
6.5.3 A Validation Question and Specification's Behavior Space 108
6.5.4 Approximation and Reformulation 109
6.5.5 Related Works 110
6.5.6 Evaluation of ASC 113

7 Evolution **115**

7.1 Why evolution occurs 115
7.2 Evolution transformations—support for evolution 117
7.3 Advances in ARIES with respect to evolution 118
7.3.1 Recap: the state of evolution transformations in the Specification Assistant 118
7.3.2 ARIES advances 118
7.4 Semantic properties and effects 120
7.4.1 Dimensions of semantic properties 120
7.4.2 Generic network modification operations 121
7.4.3 Examples of dimensions of semantic properties and changes within them 122
7.5 Transformations, effects and semantic dimensions 124

7.5.1 Linking manipulations within presentations to effects 124
7.5.2 Linking evolution transformations to effects 126
7.6 Related work 127
7.7 Examples of evolution transformations 128

8 Future Directions **132**
8.1 Improved acquisition and presentation modes 132
8.1.1 Domain-specific acquisition and presentation 133
8.1.2 Acquisition using demonstration examples 133
8.2 Representation issues 133
8.2.1 Modularity for information flow diagrams 134
8.2.2 Modularity for state transition diagrams 134
8.3 Support for cooperation and reuse 136
8.3.1 Merging workproducts and other CSCW support 136
8.3.2 Reuse constructions and retrieval 137
8.3.3 Folder structuring and heterogeneous knowledge representations . . 137
8.4 Additional intelligent assistance 138
8.4.1 Formalism for the specification evolution process 139
8.4.2 Guidance for non-experts 139
8.5 Evaluation 139

9 Acknowledgements **140**

List of Figures

1.1 Overburdened software engineers 2

1.2 Analysts using ARIES 3

1.3 Physical decomposition hierarchy starting from aas 18

1.4 Functional decomposition hierarchy for accc 19

2.1 Review and modification of requirements 25

2.2 Overall view of the ARIES interface 26

2.3 Overview of the handoff folder 28

2.4 Event taxonomy for init-handoff 29

2.5 English paraphrase of the init-handoff event 29

2.6 Parameter menu for transformation 31

2.7 New English paraphrase of init-handoff 32

2.8 English paraphrase of modified manual-init-handoff 32

2.9 State-transition view of handoff enablement 33

2.10 Reusable Gist view of enter-enabled-handoff 34

3.1 Taxonomy of kinds of motion 37

3.2 An RG presentation of an invariant 38

3.3 Definitions of takeoff and move 39

4.1 Constructing a presentation 52

4.2 Spreadsheet presentation of nonfunctional requirements 56
4.3 Translation from the internal representation to RG 59
4.4 Definition of Event Taxonomy presentation 61
4.5 ARIES Process Model presentation 64
4.6 An instructional script 69

5.1 Taxonomy of kinds of motion 73
5.2 The aircraft folder 73
5.3 Typical relationship among folders 75
5.4 Folders containing several models for the "direction" concept 77
5.5 Specialization hierarchy of direction folders 78
5.6 Folders used by automatic-tracking-capability 81
5.7 Updated use presentation 81
5.8 Final state of the use list for automatic-tracking-capability 82
5.9 Reusable Gist definition of handoff 83
5.10 Definition of init-handoff in handoff folder 84
5.11 Definition of init-handoff in the automatic-tracking-capability folder 84

6.1 A Successful Handoff Scenario for Validation Question VQ1 101
6.2 A Handoff Anti-Scenario for Validation Question VQ2 102
6.3 Reusable Gist definition of the event *Accept-Handoff* 104
6.4 Paraphrase of the event *Accept-Handoff* 105
6.5 Primitive Influence Graph of the event *accept-handoff* 106
6.6 Influence Graph of the event *accept-handoff* 107

7.1 Menu of modifications to the event-taxonomy presentation 124
7.2 Menu of evolution transformations retrieved in response to selecting Create event-declaration node 125

7.3 An example of an evolution transformation 130
7.4 A taxonomy of transformations 131

Chapter 1

Overview and Motivation

In the Requirements/Specification Facet for KBSA project, we have explored issues and built plausible models in the field of computer support for the first half of the software development life cycle. We have built an experimental requirements/specification environment called ARIES [1] which requirements analysts may use to codify system requirements in a manner that enables extensive machine support for evaluation and reuse. Many ARIES components have reached a mature level. Others lay the foundations for advances to come. In total, the concepts uncovered in our research and the capabilities of the existing prototype will be of interest to the reader who is investigating the following questions.

- What lies ahead for Computer-Aided Software Engineering (CASE) tools?
- What are the key issues for automating the software development process?
- Is there value-added in knowledge-based approaches to software engineering?
- Is there value-added in knowledge-based approaches to formal specification development?
- What is really to be gained from having wide-spectrum (informal to formal, abstract to concrete, application domain to software constructs) representations on-line?

We see the core challenge as being the analysts' need to deal with the *large volume and wide diversity of knowledge* associated with the requirements engineering process. Figure 1.1 illustrates the problem. This knowledge includes domain models, initial requirement conceptions, abstracted views of requirements, formal descriptions of systems, and stereotypical ways to modify these descriptions. ARIES embodies our ideas on how to support analysts to deal effectively with such knowledge. Its features include:

[1] ARIES stands for Acquisition of Requirements and Incremental Evolution of Specifications.

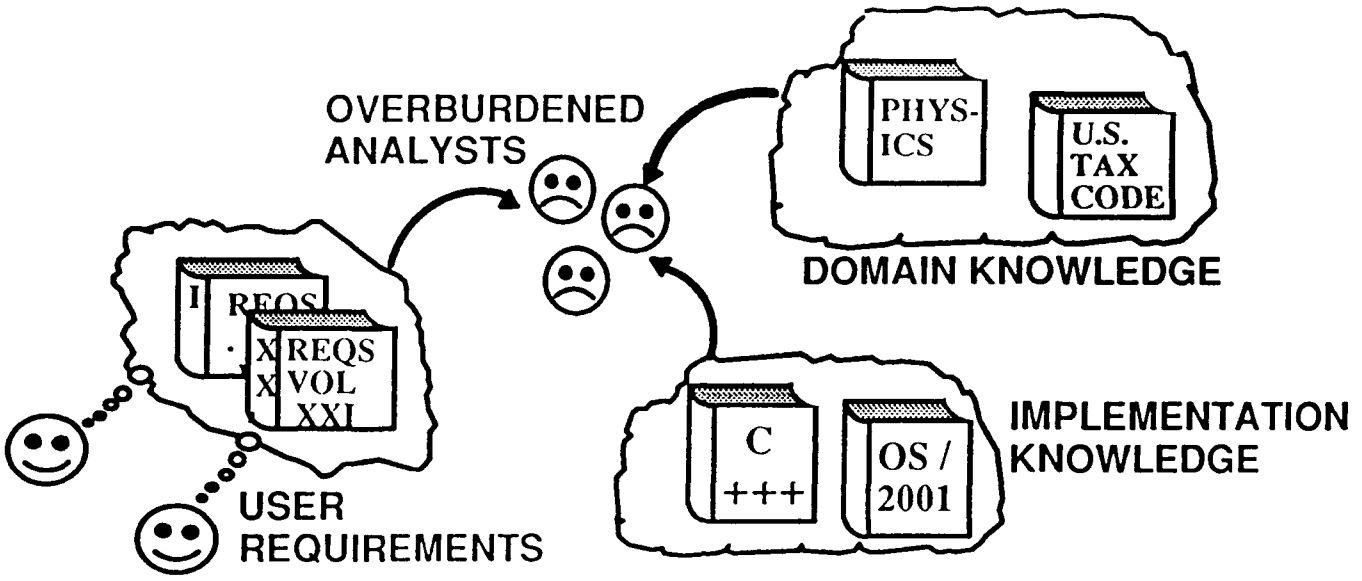

Figure 1.1: Overburdened software engineers

- a modularized central repository for requirements information,
- a single, highly expressive internal knowledge representation scheme,
- communication support between analysts and the system in terms and styles familiar to the analyst, and
- tool support for the analyst to manage, analyze and evolve requirements information.

Together these features significantly reduce the burden on requirements engineers, enabling them to perform their tasks more productively and reliably.

ARIES research has addressed several roadblocks in providing automated assistance to the process of developing formal specifications. One of the principal roadblocks is that there is a large gap between initial requirement conceptions and more formal machine-manageable descriptions. ARIES provides tools for the gradual evolution of acquired requirements, expressed in hypertext and graphical diagrams, into formal specifications. These tools also alleviate another difficulty with specifications, that it is hard to modify them to reflect changing understandings of requirements, without inadvertently introducing errors. ARIES is particularly concerned with problems that arise in the development of specifications of large systems. Specification reuse is a major concern, so that large specifications do not have to be written from scratch. Mechanisms are provided for dealing with conflicts in requirements, especially those arising when groups of analysts work together. Validation

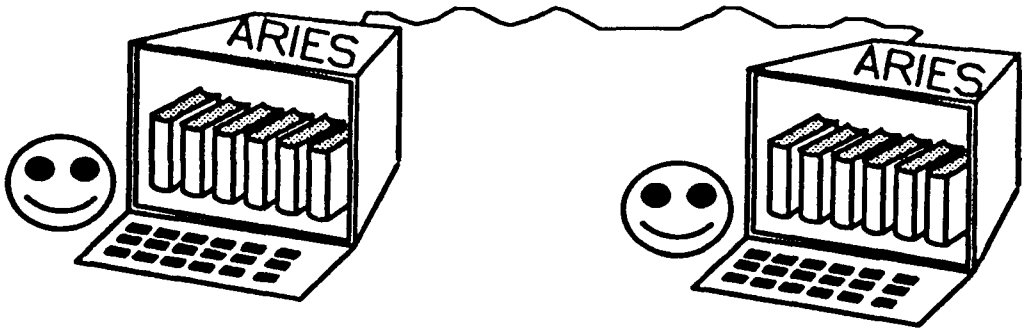

Figure 1.2: Analysts using ARIES

techniques, including simulation, deduction, and abstraction, are provided, to cope with the problem that large specifications are difficult to understand and reason about.

From an artificial intelligence perspective, this work is significant in that it deals with the representation and presentation of large knowledge bases and the needs for effective knowledge segmentation so that multiple tools can operate on descriptions at different levels of abstraction and so that users can effectively communicate with an intelligent assistant program. Requirements acquisition is a special case of knowledge acquisition, and many of the techniques in ARIES to support requirements acquisition are of interest from a knowledge acquisition perspective as well. Examples of such techniques include transformations for changing domain models, and support for the construction of task-specific domain models from reusable components.

From a software engineering perspective, this work formalizes software engineering artifacts and processes so that automated support for software development can be built on a strong semantic foundation.

This report is organized as follows. The remainder of this chapter provides motivation for the ARIES work. It identifies shortcomings in conventional CASE approaches, and highlights the issues which we feel an advanced requirements engineering support environment should address. It then outlines the capabilities that have been built into ARIES to help address these concerns. Chapter 2 presents an example scenario of ARIES in use, illustrating a number of ARIES's capabilities and showing how they support the requirements analysis process. Chapter 3 discusses the issues that must be addressed in order to build a representation of requirements knowledge that supports such capabilities, and describes how these issues are attacked in ARIES. Chapter 4 discusses how presentations are supported. Chapter 5 describes the techniques in ARIES for supporting cooperation between analysis when developing requirements, and for supporting reuse of previously formalized requirements. Chapter 6 discusses the approaches which have been taken in ARIES to rea-

soning about specifications. Chapter 7 describes the mechanisms for supporting evolution in ARIES. Chapter 8 identifies areas of potential future investigation, building upon the accomplishments so far.

1.1 Specification development for large complex systems

Our design decisions for ARIES follow from observations on the requirements/specification process and on the modes of analyst/ARIES interaction necessary for effective automation support. We draw no distinction between "requirements" and "specifications" of systems. Rather, we envision a seamless requirements/specification process resulting in a software requirements specification (SRS) which describes a system to be built at many different levels of detail or formality. This position echoes the position taken by others working in requirements analysis, such as Davis [17].

Davis, in his book *Software Requirements Analysis and Specification* [17], identifies a number of properties that an SRS should have:

- it should be *correct*—the requirements that it contains are in fact required of the system to be built;
- it should be *unambiguous*, so that each requirement may be interpreted in only one way;
- it should be *complete,* describing all properties that the system should have;
- it should be *verifiable*—there is some cost-effective means for testing whether the requirement is met;
- it should be *consistent*—no subset of the requirements conflict;
- it should be *understandable by non-computer specialists*;
- it should be *modifiable*—necessary changes to requirements may be made easily, completely, and consistently;
- it should be *traceable*—each requirement may be traced to its origin in other documents, and may be traced to the software component(s) satisfying the requirement; and finally,
- it should be *annotated*—each requirement should be marked to indicate how necessary and volatile it is.

Clearly these properties are difficult to attain at the same time. Nevertheless, it is useful to think of requirements analysis as attempting to gradually achieve them to the greatest extent possible.

From the perspective of computer-based support, the focus needs to be as much on supporting attempts to gradually achieve these goals as it is on determining if a given specification meets the goals as stated. With this in mind, we turn to a review of how SRS goals are being being met today and how they might be met in the future with increasingly sophisticated software tool support in the hands of large teams of analysts.

1.2 The state-of-the-practice

The ARIES effort grows out of and reacts to the essentially informal practices in requirements management that exist in the work place today. This discussion of the state-of-the-practice of requirements management covers several areas. We begin by pointing out the crucial importance of requirements specification within the Department of Defense (DOD) community. Next we look at two emerging trends—the role of reuse and the focus on process improvement (specifically, the impact of Software Engineering Institute (SEI) contractor assessment). Finally, we examine the requirements management functionality available in commercial tools, and some deficiencies in this functionality.

For DOD sponsoring agencies, requirements specification is an integral part of software procurement. In preparing a Request For Proposal (RFP), the responsible agency states precisely what functionality is needed for the proposed system. Within this agency, a designated group of analysts—having expertise in writing requirements specifications—converses with the users of the proposed system in order to gain an understanding of user needs. Then the requirements specification analysts take the users' input and transforms it into a requirements document appropriate for directing the bidding contractors. A clear, exact, and understandable statement of the needed functionality is necessary in order to correctly convey to the bidding contractors what it is that the contracting agency wants built, thereby enabling the bidding contractors to accurately develop a proposal that addresses the users' needs.

After contract award, one of the first products developed by the winning contractor is the SRS which documents the results of the software requirements analysis phase. Many programs are developed in accordance with DO-STD-2167A, which specifies the format and content of the SRS. This standard requires that an SRS identify the external interfaces, the capability requirements, the internal interfaces between those capabilities, and trace requirements between the SRS and the sponsor-produced document. This SRS permits the sponsor to get a clear understanding of exactly what the system will be built to do

and is the basis of achieving agreement between the contractor and the sponsor. After agreement, further requirements changes from the sponsor are "out of scope" and may result in costly re-negotiation. In addition, the SRS is a key document for the contractor during the subsequent development phases. Stakeholders use the SRS to trace requirements to the design and code to assure that the resulting system satisfies the requirements. During system test, the software test organization tests the final system against the SRS to guarantee compliance. Questions from the sponsor during acceptance testing are resolved by referring to the agreed upon SRS.

An important area of interest within the DOD community is the reuse of software artifacts. In the past, the reuse emphasis has been on informally reusing code from a previous project. This usually failed because the old code didn't fit the new architecture. What is needed is a process that involves planning and engineering for software families rather than single systems. This includes an analysis of the domain to identify the common and variable parts, and a reusable requirements specification for the family. Even an informal reusable requirements document would be very helpful to analysts. Such a document would describe the common parts of the family of systems and separately describe the variable parts. To reuse this specification, the analyst would extract the common part as well as any variation that would describe the particular instance being developed. If no variation was currently identified that applied to this instance, the analyst would write the needed variable part and add it to the reusable specification, thereby broading its description of the family. This reusable requirements specification is valuable at all stages of software development because analysts can use it to understand which portions of the system are reusable and which portions will require modification.

A sampling of current activities in the reuse area include:

- At the San Antonio I workshop, the fifth in a continuing series of Joint Logistics Commanders sponsored government and industry workshops focusing on identifying DOD software problems and solutions, recommendations were made for instituting reuse. These recommendations were that a software application be viewed as a member of a family of systems, domain analysis be employed, and guidelines be established for creating reuse assets. As a result, a Software Reuse Subgroup was formed. This group's initial activities are to develop a guidebook for reuse, organize a domain analysis workshop, and identify approaches and barriers to reuse.

- In July, 1990, the Defense Advanced Research Projects Agency (DARPA) and the SEI Software Architectures Engineering Project hosted a workshop on domain-specific software architectures.[2] The participants presented position papers on how model-based, common architectures can improve the development of large-scale defense

[2]This meeting was reported in the Software Engineering Institute's publication *Bridge*, Sept. 1990.

systems. Addressing generality and the reduction of complexity in application areas will allow the development and reuse of common structures across those areas.

- The Software Productivity Consortium (SPC), an aerospace industry initiative, is seeking to bring quality and productivity improvements to software development practice. A Reuse Maturity Technical Advisory Group has been established to advance the processes, methods, and tools involved with software reuse. A reuse library tool has been developed to support classification, storage, and retrieval of reuse assets. Pilot projects are being pursued in domain analysis and the development of a reusable requirements specification. The importance of these efforts is revealed by the report by a member company that they have lost contracts to competitors with established reuse libraries and domain specific software architectures in their proposals.

One further item that needs to be addressed in the area of reuse is the cost of the effort. Significant resources are needed to do a domain analysis and to write the initial reusable requirements specification. Companies are reluctant to make this investment. However, successes by those companies willing to forge ahead will illustrate how beneficial this investment can be.

Another important emerging theme is the SEI initiative for standardizing and improving the software development process. This initiative, sponsored by the DOD, aims to improve the entire software process. The SEI has developed a Capability Maturity Model to rate the maturity of a software process. This model consists of five levels, level one being the initial level and level five being the optimal, mature level. The initiative offers some specific objectives for improved requirements management—an area that often receives limited attention in practice. The current maturity model requires that contractors put mechanisms in place for controlling changes to the software requirements, for ensuring that the software design teams understand each software requirement, and for ensuring traceability between the software requirements and top-level design. In the draft version of the maturity model due to be instituted in November, 1992, capabilities for requirements management are expanded and called out as a separate key process area for achieving level two. Many contracting agencies are beginning to use maturity ratings as a means of assessing contractor qualifications. For example, requests for proposals may state that the bidding contractor must be at a maturity level of three or be considered high risk, or that the bidding contractor must be at a maturity level of two with defined plans to reach level three. This initiative can be expected to create an increased awareness of requirements level issues within the DOD community.

Given the importance of requirements management, commercial companies have provided CASE tools to assist in performing software requirements analysis and software requirements traceability. Before the advent of CASE technology in the mid-1980's, analysts wrote an

SRS directly in text format with little coupling to background engineering analysis. These documents frequently did not include needed detail and could be ambiguous. With the use of CASE, the resulting SRS contains more detail and is more precise. There are numerous CASE products that are hosted on a wide range of platforms. However, these products tend to have the following core features in common.

- An Entity Relationship Diagram is used to show information modeling at a high level of abstraction.
- A Context Diagram is defined to show the system and all its external interfaces.
- Data flow diagrams are used to define the software functionality, and the processes on the data flow diagram are decomposed to a low level of detail.
- Mini-specifications provide a textual description of the execution of the lowest level processes on the data flow diagrams.
- All data elements are identified down to their primitive form (units, range, accuracy) and are retained in a data dictionary. This single data dictionary assures consistency among terms used by the analysts working on different sections of the SRS.
- The flow of control and various system states are illustrated by finite state machines, e.g. State Transition Diagrams, State Event Matrices.
- Methodology and consistency checking of the diagrams and data elements can be requested by the user.
- Requirements traceability is provided by the ability to associate requirements with the entities in the software decomposition.
- Templates of MIL-STD SRSs are provided and the information from the database is inserted to create the finished document.

Defining a system to precise detail does result in more time spent in the software requirements analysis phase of the program when compared to the conventional practice. However, the software development team will iron out many details early in the process, thus saving time later in the development lifecycle. The sponsor will receive a more detailed specification, thereby getting a better picture of what will be built. And the software test organization will be able to test the system against a detailed document, thereby enhancing testing accuracy.

When investigating technology developments in all software arenas, it must be asked whether the technology is sufficient to support a particular computational paradigm. With

CASE technology, deficiencies in functionality or tedious and frustrating procedures can impact a program's schedule and wipe out any benefits gained from using a machine managed approach. These deficiencies in the existing tools fall into three categories: limited expressive power, lack of automatic reasoning, and limitations on obtaining informative views of what has been specified.

The current CASE tools lack the expressive power required for capturing high level specifications in general and for dealing with reuse issues in particular. For example, a process on a data flow diagram must be decomposed into one and only one child process or minispecification. There is no capability to define an alternate decomposition if a certain condition is true. Therefore for a family of systems, the common part can be defined. But there is no way to tie the other variations into the same model and be able to select the common part and the desired variation(s) depending on conditions for this instance of the system. Thus the DOD thrust toward families of systems cannot be supported by current CASE technology.

Another deficiency with the current CASE tools is their lack of automatic reasoning. When a requirement changes, updates must be done manually in all areas that are affected by the change. Also there is no verification that if one condition is true, then all conditions that depend on it are in agreement. The analyst must verify that this agreement exists by reviewing all the conditions. This tedious process turns into a major effort for specifications for large systems.

Lastly, there are limitations in producing informative views of specifications. In the area of requirements traceability, the current CASE tools generate reports, but frequently the reports do not include requirements traceability. Sponsors need this information to assure that indeed their requirements have been adhered to in the specification work. The typical format for this information is a requirements traceability matrix indicating the flow of requirements from the government specification to the SRS and the Software Design Document. The matrix is used to verify that the system built complies with the system requested by the sponsor and agreed to by the contractor, and the software test organization uses this matrix to ensure that compliance during component and system test. In addition, the production of textual documents with the current CASE tools is often a significant effort. Many of the tools use an external publishing system to produce publication quality documents. This process can be very lengthly for a substantial document and frequently engineers must make formatting corrections manually. For example, text must be resized to fit within the confines of a table. While this does not limit the functionality of the CASE tool, it does have a substantial impact on a program's schedule.

1.3 The ARIES paradigm—requirements acquisition and specification evolution

The ARIES paradigm consists of a broad conception of the requirements/specification process and a commitment to a specific technology base for achieving substantial computer assistance.

1.3.1 Assumptions about the process

ARIES assumes a model of software development in which there are multiple goals for requirements analysis. The process we outline may differ from abstract models of software development, but we feel it matches the actual practice of today's engineers and suggests how that practice will evolve with the inclusion of more and better software tool support. Requirements analysis produces a software requirements specification (SRS), describing the characteristics of the system to be built. However, such documents are themselves but a means to achieve a more fundamental goal, namely communication of requirements to designers and stakeholders (end-users, procurement agents, etc.). In fact engineering media—diagrams, outlines—used along the way toward producing a written document can be extremely informative. Executable prototypes are another useful product, both to help communicate requirements and to validate the accuracy of those requirements. Finally, we assume that system requirements are not developed from scratch and thrown away. Instead, a goal of the requirements analysis process should be to reuse requirements from other systems or classes of systems, where appropriate, modifying these requirements as necessary. Likewise, we assume that analysts have some desire to organize their requirements specifications in such a way that they might be reusable on future projects.

Typically groups of many individuals, representing differing specialties and working at different levels of abstraction, perform the requirements engineering task. The work proceeds through diverging and converging attempts at domain modeling, working toward developing a shared vocabulary to be used for describing requirements, and through extensive exploration of specification development issues along multiple paths.

One can roughly break the process into three phases—acquisition, evolution, and analysis. An acquisition process incorporates new information about a system from analysts, who may collect this information from a variety of sources, including interviews with clients and source documents describing the domain in which the system will operate. An evolution process, directed by analysts, reformulates statements about a system so that they are more precise, formal, and/or implementable. It also corrects inaccuracies in requirements, and revises requirements that cannot be satisfied strictly as stated. An analysis process

explores consequences of requirements statements, looking for inconsistencies.

1.3.2 The approach to automation

The section above describes a process that could be performed entirely manually or with the aid of knowledge-based assistance.

There are many approaches to providing machine support for requirements/specification work. The degree of automation is one important dimension. At one extreme are expert system approaches which attempt full automation of some part of the process. At the other extreme are systems which provide very limited support—you can say anything with little consistency checking, no propagation is possible because no underlying formal model exists. We are aiming in the middle at intelligent assistance. In ARIES, facilities operate on the system knowledge base, a repository of knowledge about stereotypical domains, reusable requirements/specification components, and descriptions of specific systems being developed. Generic knowledge and individual system specifications are represented in the same manner, and tools apply equally to both. We expect that the analyst will make all the critical decisions while the machine will help to put the pieces of the system together—through knowledge of artifacts and typical processes.

1.3.3 The output of the process

The output from this ARIES process consists of two parts: an implementable specification, which in an automatic programming setting serves as input to a mechanical optimization process, and on-line requirements communication vehicles—documents, diagrams, and simulations—which describe the system to be built in precise terms. These two outputs subsume the role of SRS's in conventional software development.

1.4 History of the ARIES Project

1.4.1 The KBSA Report—short term and long term goals

ARIES is a product of the ongoing Knowledge-Based Software Assistant (KBSA) program. KBSA, as proposed in the 1983 report by the US Air Force's Rome Laboratories [30], was conceived as an integrated knowledge-based system to support all aspects of the software life cycle. Such an assistant would support specification-based software development: programs would not be written in conventional programming languages, but instead would

be written in an executable specification language, from which efficient implementations would be mechanically derived. First, the process of requirements analysis generates a formal specification from informal requirements. The formal specification then initializes a mechanical transformation process, yielding an optimized program. Analysts validate specifications through examination and analysis of a formal specification, not by testing the optimized code. Analysts maintain programs by modifying the specifications, and deriving new optimized programs. In a complete KBSA system, and to some extent in ARIES as well, requirements analysis tools and implementation tools are integrated into a single environment, allowing analysts to perform exploratory prototyping during requirements analysis. Although ARIES is part of KBSA, the design of ARIES does not preclude its use in situations where a full KBSA system is not available.

The original KBSA report called for extensive work aimed at each of several phases of software development. For each phase a list of goals were advocated to direct the efforts. The ARIES effort builds on the results of two earlier efforts addressing those goals. At Lockheed Sanders, we built the Knowledge-Based Requirements Assistant (KBRA) [37]. At USC/ISI, specification construction, validation, and evolution goals were addressed in the Knowledge-Based Specification Assistant [65, 45, 44]. We briefly describe each of these systems in the next paragraphs.

1.4.2 The Requirements Assistant

The KBSA long-range goals for the requirements facet were to provide "comprehensive requirements management, intelligent editing of requirements, testing of requirements for completeness and consistency (both self-consistency and consistency with application domain models), performing requirements reviews, maintaining and transforming requirements in response to changes, decomposing and refining requirements into executable specification languages, and acquiring requirements knowledge." [30]

The KBRA prototype realized many of these goals. Facilities for acquisition of informal requirements, entered as structured text and diagrams, were developed. KBRA's limited case-frame-based natural language understanding ability assisted in the formalization of informal text by recognizing words in a lexicon of domain concepts. KBRA allowed analysts to describe systems from different points of view, e.g., data flow, state transition, functional decomposition, spreadsheets of nonfunctional properties, and text outlines. An internal representation of the system being built integrated these different views. It was also able to generate an SRS document from the system descriptions acquired through the multiple acquisition modes.

1.4.3 The Specification Assistant

The KBSA Report's goals for the specification facet were to manage the first formal representation of the system to be built. Specifications were to be validated, used as a testable prototype, and used as the starting point for source-to-source program transformations which would be used to convert the specification into an efficient implementation.

The principal contribution of the Knowledge-Based Specification Assistant was the development of *evolution transformations* for specification modification [46]. Evolution transformations are operators that modify system descriptions in a controlled fashion, affecting some aspects of a requirements statement while retaining others unchanged. They also propagate changes throughout a system description. Significant effort was invested in identifying evolution steps (both meaning-preserving and non-meaning-preserving) that routinely occur in the specification development process, and automating them in the form of evolution transformations. The Specification Assistant also provided validation tools in the form of a paraphraser which translates specifications into English [75, 58], a symbolic evaluator for simulating the specification and proving theorems about it [14], and static analysis tools which automatically maintain and update analysis information as the specification is transformed [47].

1.4.4 Motivation for the ARIES project

These early efforts intersected in several unanticipated and informative ways—pointing out the interconnection between requirements and specification concerns and setting the stage for the ARIES project. Tools that were developed for one phase often could be applied in the other phase as well. As illustration, analysts might wish to use KBRA's data flow diagram presentation during the end stages of specification development. Inversely, analysts would clearly like to have specification assistant simulation capabilities available at the earliest stages of requirements acquisition. Importantly, we recognized the need to provide for a smooth transition between informal acquisition modes (an "electronic white board" metaphor) and formal manipulation modes (evolution transformations applied to a formal language). Overall, we concluded that if the capabilities of the Requirements Assistant and the Specification Assistant were integrated into a common framework, the resulting system would provide more effective support than either system did by itself.

Performing such an integration, however, involved solving some technical problems that neither precursor system had addressed. For example, although the KBRA supported multiple notations, the semantics of these notations did not overlap; furthermore, the expressiveness of the KBRA's notations was limited. Integration with the Specification Assistant forced us to support additional notations with overlapping semantics, some of which are

highly expressive. This led to significant extensions both to the underlying representation framework and to the presentation system. Other technical problems arose as we extended ARIES to more complex problems, so that the scalability of ARIES technology could be demonstrated. Issues of knowledge organization, sharing, and reuse became increasingly critical, as did issues of how to support groups of analysts working together on a project. Our work has by no means solved all of these problems, but significant progress has been made.

1.5 Project approach

Our approach has been to use knowledge-based technology in general, use a transformational paradigm in particular, and to direct the research by applying it to large complex domains.

1.5.1 Knowledge-based technology

Our approach has been to represent requirements/specification concerns in a single common knowledge base. Individual units of system description are treated as examples of general concepts and are subject to underlying reasoning mechanisms which propagate information and check for consistency as an analyst evolves a specification. The basic units of system descriptions in ARIES are *types*, *instances*, *relations*, *events*, and *invariants*. Airports, radars, aircraft, airspace are examples of types in the air traffic control domain. Logan and Los Angeles International are examples of instances of airports. The control relation models an important aspect of this domain. It connects controllers to aircraft in the airspace. Events include actions of agents such as the move event. Invariants establish restrictions on behavior. For example, an invariant might express what happens to the control relation when an aircraft moves between sectors of the airspace. Any of the illustrated examples can be described in many different ways—informal or formal, abstract or concrete. In order to ensure consistency in the underlying reasoning, we have chosen to represent all such descriptions in one common way. Details about the different unit types are presented in Chapter 3

1.5.2 Transformation technology

Transformations are encoded knowledge about the process of software engineering. Evolution transformations are the principal mechanism for supporting evolution in ARIES. They are operators that modify system descriptions in a controlled fashion, affecting some

aspects of a requirements statement while retaining others unchanged. In selecting a transformation from a library of transformations, an analyst modifies previous work at a level significantly above error-prone text editing. The transformation checks the specification to ensure that the change is consistent with other decisions and automatically propagates changes throughout the specification. Transformations are represented declaratively in the knowledge base as a special kind of event; this means that tools for describing and presenting events can also be used to describe and present transformations.

1.6 Use of real world examples

While our approach to requirements analysis aims to broadly support requirements acquisition and analysis, we have focused our efforts by working with detailed specific application domains. In past efforts we had concentrated on several relatively narrow domains, including hospital patient monitoring, library systems, signal processing, and tracking. Then under the Requirements Assistant effort, we started an initiative in requirements for air traffic control systems. We conducted extensive interviews with air traffic control system engineers; an engineer created an annotated engineering notebook that we used as reference material; and the KBSA community adopted air traffic control as a common application domain. Subsequently this domain was used to drive the development of the precursor Requirements Assistant and Specification Assistant prototypes.

Our investigation under ARIES has proceeded in three phases. In the first phase, we recast and extended earlier work on the air traffic control domain. In a second phase, in the spring of 1990 ARIES project members performed an experimental exercise in the road traffic control domain. We felt that this domain would be small enough that it would be possible to analyze from beginning to end in a limited amount of time. This study provided insights into how knowledge-based tools could be employed to coordinate groups of requirements analysts. It also suggested ways of organizing domain and requirements knowledge in a reusable manner, so that the same components could be used both for air traffic control and road traffic control. Finally, we returned to the air traffic control domain, to flesh out earlier efforts based on many of the insights derived from the smaller road traffic control study.

The following two sections draw from these experiments. First, we describe some the general issues that are associated with road traffic control specification; then we give a detailed analysis of evolution in the broader air traffic control domain.

1.6.1 Developing a road traffic control specification

In the road traffic control exercise, each project member worked independently on one aspect of the road traffic control problem. We then compared notes to find common themes and to compare this work with the body of air traffic control requirements that we had already formalized. The following are some of the issues that different project members investigated in analyzing the road traffic control problem:

- understanding what restrictions must be placed on the duration of traffic light signals to ensure safe and expeditious flow of traffic,
- identifying requirements by viewing the problem as an instance of a generic scheduling problem, and determining what general requirements of scheduling problems apply to road traffic control,
- identifying possible states of the traffic lights, and conditions under which the system changes state, and
- sketching possible algorithms for coordinating lights.

There were other issues that were not explicitly raised, but which were implicit in a number of these investigations, such as how the traffic lights, the traffic light controller, and traffic sensors would be connected to each other.

One place where the tension between coordinated work and independent work was strong was in modeling the application domain. It was readily apparent that everyone had a similar intuitive model of the road traffic control domain. This included concepts such as vehicles, roads, colors, and directions. There were also common notions of system components, including traffic lights and roadbed sensors. At the same time, there were key differences in domain models, depending upon what task each analyst was performing. For example, two distinct models of vehicle motion arose. In one, vehicles appear at the entrance to the intersection, traverse the intersection, and then disappear. This corresponds roughly to the information that a traffic light system has about the environment solely on the basis of what road sensors can provide. In another model, vehicles have a distance from the intersection, a velocity, and an acceleration, and approach and depart from the intersection in a continuous process. This latter model was needed to understand the requirements imposed on a traffic light system because of vehicle behavior (e.g., how much time must be allowed between light changes). We needed a way to support such conflicting models, and at the same time understand how requirements stated in terms of one model might be reformulated in terms of another model.

The road traffic control analysis made use of several notations—text, diagrams. No one notation would have been adequate by itself. The primary notations used were natural language, state transition diagrams, entity-relationship-attribute notations, and mathematical constraints. People would sometimes describe requirements first in natural language, and then write formal statements to capture the meaning of the natural language. In order for other people to understand these formal statements, traces back to the original natural language were extremely important.

Simulation and execution were useful for getting the requirements right. Requirements statements that seemed reasonable at first in fact permitted anomalous behavior, such as traffic lights changing unnecessarily. Simulation made it possible to understand the dynamics of the domain properly; for example, a simulation of traffic flow through the intersection helped to determine how long it takes for traffic waiting at an intersection to resume normal flow.

1.6.2 Developing an air traffic control specification

To demonstrate the power of the ARIES approach, and its ability to handle large complex specification problems, we have devoted significant effort to a single domain, namely air traffic control. We have used two sources for this work. First, we have been modeling requirements for a particular system—the control system used for air traffic control in the airspace around Tempelhof Airport in Berlin. Second, we have studied the requirements for U.S. domestic en route air traffic control systems, i.e., those systems responsible for the control of air traffic cruising at the high altitudes reserved for jet aircraft. These requirements are drawn from manuals on flight procedures (e.g., [4]), and from the experiences of the Federal Aviation Administration's Advanced Automation Program [41], whose goal is to develop the next generation of air traffic control systems. We have also interviewed requirements analysts in this domain, and information processing specialists with the current air traffic control automation system.

This domain pointed out the need for tools which help manage the diversity of components associated with a system description. The Federal Aviation Administration's Advanced Automation System (AAS) actually includes descriptions of multiple systems. First of all, analysts distinguish between the intermediate stage computer facility that will be delivered in the near term and the ultimate AAS, which will support air traffic control into the next century. The intermediate delivery will consist of new automation support for controllers within the existing organization of air traffic control facilities. The ultimate delivery will support a new organization of the entire air traffic control system, in which a new kind of air traffic control center called an air control facility (ACF) is introduced which takes over the functions of a combination of centers in the current air traffic control organization.

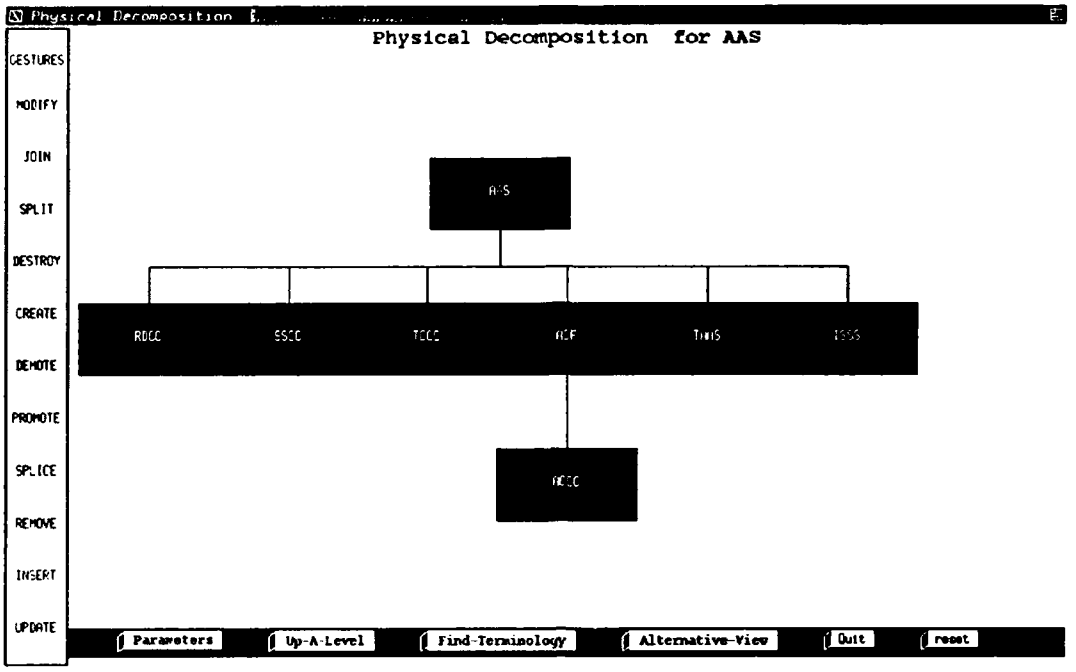

Figure 1.3: Physical decomposition hierarchy starting from aas

Even from this simple overview, the need for engineers to decompose complex objects (systems, functions, data) into manageable-sized components becomes very apparent. Figures 1.3 and 1.4 show some of the project-specific decompositions that are important. [3] We have captured these decompositions in ARIES constructs called *folders*. Details of folder concepts and implementation are taken up in later sections of this report. Here we informally illustrate a few air traffic control folders to demonstrate the need for segmentation and organization of complex projects.

The uppermost folder in this tree is called aas; it includes concepts of the AAS program which apply to any system defined as part of that program. We developed separate folders with descriptions of the intermediate-stage system (ISSS) and the computer system to be used in an Area Control Facility (ACF). These folders are named isss and acf, respectively. One part of the Area Control Facility is the Area Control Computer Complex (ACCC), the hardware-software configuration responsible for operational control of air traffic in a given sector.

[3] In order to give an accurate picture of the use of ARIES, we will make extensive use of screen images of ARIES. However, the reader should be aware that ARIES is designed to be used on a color monitor. The images are somewhat more difficult to read in black and white, because important color distinctions are missing. The problem is evident in this figure: the boxes in the diagram appear to run together, whereas on a color display the borders of each box can be easily seen.

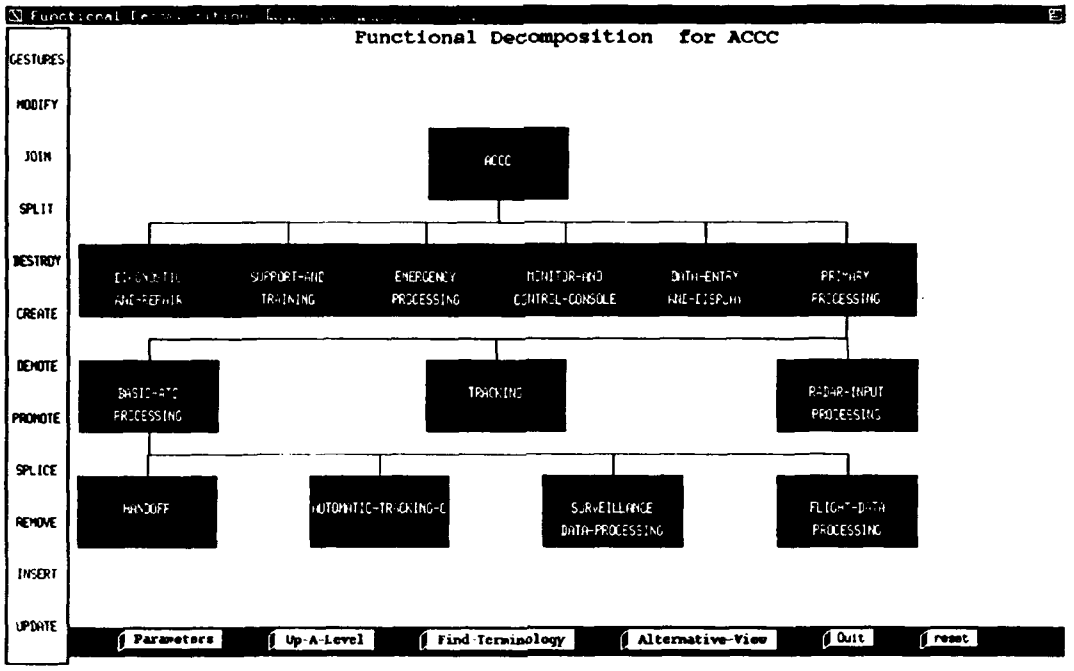

Figure 1.4: Functional decomposition hierarchy for accc

Other folders are used to organize the different functional areas of the individual systems being described. Figure 1.4 shows the various functional decomposition folders for the Area Control Computer Complex. At the lowest level of detail, we have developed folders for particular functional areas: handoff, flight-data-processing, traffic-management-capabilities, automatic-tracking-capability, and surveillance-data-processing.

1.6.3 Lessons learned

The above experiments led to a number of conclusions regarding the nature of the specification development process. Several central issues relating to specification development were identified: coordinating multiple users and viewpoints, capturing requirements that can be shared across systems, and sharing core concepts and knowledge across domains.

1.6.3.1 Multiple viewpoints must be coordinated

The requirements for future air traffic control systems are extremely detailed: system descriptions for the Advanced Automation System (AAS) run into the hundreds of pages. The work of specification must be divided among multiple analysts in order to be feasible. In

our current analysis and formalization of sections of the AAS requirements, we find that the FAA has identified particular functional areas for that system. These areas, including track processing, flight plan processing, and traffic management, seem to be good candidates for assignment to different analysts or analyst teams.

However, one important conclusion we have drawn is that a proper balance must be struck between coordinated and independent work of analysts. Requirements are not like program modules, that can be handed off to independent coders to implement. There is inevitably significant overlap between them. They may share a significant amount of common terminology between them. Requirements expressed in one functional area may have impact on other functional areas. In the AAS specification, we specified track processing, flight plan processing, and assignment of control separately. By comparing notes, we then found that flight plan information had an impact on how tracks are disambiguated, and that the process of handing off control of aircraft from one facility to the next had an impact on when flight plan information is communicated between facility computer systems. Our approach to this issue has been to work on machine-mediated ways to support separation and subsequent merging of work products, rather than to force analysts to constantly coordinate whenever an area of potential common concern is identified.

1.6.3.2 Inconsistency is pervasive

Separate development of different requirements areas inevitably leads to inconsistencies. These inconsistencies are a natural consequence of allowing analysts to focus on different concerns individually. Although consistency is an important goal for the requirements process to achieve, we have concluded that it cannot be guaranteed and maintained throughout the requirements analysis process without forcing analysts to constantly compare their requirements descriptions against each other. Therefore, consistency must be achieved gradually, at an appropriate point in the specification development process. Nevertheless, it may not be possible to recognize all inconsistencies within a system description.

1.6.3.3 Multiple models must be supported

Multiple models cause inconsistencies. For example, when analysts specify radar processing requirements they must model the dynamics of aircraft motion to make sure that the system is able to track aircraft under normal maneuver conditions. When specifying flight plan monitoring, however, they can assume that aircraft will move in straight lines from point to point, and change direction instantaneously, since the time required for a maneuver is very short compared to the time typically spent following straight flight paths. One way of resolving such conflicts is to develop a specialization hierarchy that relates these models to common abstractions.

1.6.3.4 Sharing requirements across systems

The designer of an air traffic control system must make sure that computers and human agents can together achieve the goals of air traffic control, i.e., to ensure the safe, orderly, and expeditious flow of air traffic. How this will be done by the AAS is to some extent determined by current air traffic control practice. Thus the next generation of controller consoles are being designed to simulate on computer displays the racks of paper flight strips that controllers currently use to keep track of flights. Yet although air traffic control practice is codified in federal regulations and letters of agreement, and is thus resistant to change, the division of labor between computer and human controller is expected to change over time. The FAA anticipates that new computer systems will gradually be introduced into the new air traffic control framework over the next twenty years, taking increasing responsibility for activities that are now performed by controllers. Therefore, it is important to be able to represent the overall requirements on air traffic control, without being forced to commit to particular computer systems satisfying those requirements.

1.6.3.5 Sharing across domains

Just as there are opportunities for sharing across systems in the same domain, there are opportunities for sharing across domains. The road traffic control problem shares certain characteristics with air traffic control: both problems are concerned with the maintenance of safe, orderly, and expeditious flow of vehicular traffic. They both assume a common body of underlying concepts, such as vehicles, sensors, spatial geometry, etc. We have been endeavoring to model such concepts so that the commonalities and differences across the two domains are captured.

1.7 ARIES facilities

The key technical contributions of the ARIES effort are in the areas of presentation, reuse, reasoning, and evolution. The following chapters will explore each of these details in detail. Before doing so, we indicate why they are so essential for supporting requirements engineering.

1.7.1 Presentation tools

Presentation tools support both acquisition and review. The acquisition tools in ARIES support analysts in stating requirements as simply and directly as possible. If requirements

cannot be initially stated in a manner that is intuitive for the analyst or end-user, automation support is disconnected from real concerns and it is difficult for people to ensure that the requirements are correct. Acquisition in ARIES is accomplished by the following means. First, analysts use a structured text facility to either enter textual information found in relevant documents or to construct an on-line informal engineering notebook. To the extent that documents already exist and can be linked to subsequent formal specifications, ARIES importantly maintains traceability between such documents and the eventually completed specifications. Second, since natural language by itself is often awkward and ambiguous as a medium for stating requirements, other notations familiar to analysts are likewise supported: state transition diagrams, information flow diagrams, taxonomies, decomposition hierarchies, as well as formal specification languages. Importantly, these notations are all mapped onto a common representation of specifications internal to ARIES.

The review process in ARIES applies many of the same tools as acquisition, but in reverse. Information that analysts enter into the system in one notation may be presented by ARIES in a different notation. Our experience has pointed out that if the analyst has the opportunity to switch point of view, correctness and completeness of specifications can be more easily achieved. Most review tools have an analogue in the acquisition side. In fact, the internal presentation architecture (described below) exploits such analogies whenever possible.

1.7.2 Reuse tools

Analysts may define requirements by specializing and adapting requirements from ARIES's knowledge base of common requirements; this makes it easier for analysts to define requirements quickly and accurately. The requirement name space is managed through structures called *folders*. Folder developers capture, separate, and relate bodies of requirements information. Analysts control the extent to which these folders share information, and gradually increase the sharing as inconsistencies are reconciled. We place a heavy emphasis on codification and use of domain knowledge in requirements analysis. Although a number of researchers have identified domain modeling as a key concern (e.g., Greenspan [10]), it is given short shrift in typical practice. Requirements analysis is usually narrowly focused on describing the requirements for a single system. This is problematic if an organization is interested in introducing more than one computer system into an environment, or when the degree of computerization of an organization is expected to increase over time. We have been modeling particular domains within ARIES, and experimenting with reusing such knowledge in the engineering of requirements for multiple systems.

1.7.3 Reasoning

ARIES reasoning capabilities help analysts check for inconsistencies in proposed requirements, and explore consequences of decisions. We have developed three basic types of analysis capabilities. First, a simulation facility translates descriptions of required behavior into executable simulations. By populating simulations with interesting concrete instances of objects, an analyst can determine whether the stated requirements really guarantee satisfactory behavior. Second, deduction mechanisms propagate information through the system description, both to complete it and to detect conflicts and inconsistencies. Third, abstraction mechanisms extract simplified views of the system description. Using either simulation or visual inspection, analysts can validate abstracted views of systems more easily than they can validate full system descriptions. Furthermore, since ARIES derives an abstraction from the system description in a systematic way, the analyst can use this information to understand how conclusions drawn from the simplified view carry over to the complete system description.

1.7.4 Evolution

Evolution mechanisms are central to requirements analysis in ARIES: requirements statements are expected to evolve gradually over time. *Evolution transformations* are the principal mechanism for supporting evolution in ARIES. They are declarative representations for stereotypical change encoding the input parameters required, the applicability conditions, the effects achieved, and the rules that modify system descriptions in a controlled fashion, affecting some aspects of a requirements statement while retaining others unchanged. The rules propagate changes throughout a system description. We have invested significant effort in identifying evolution steps (both meaning-preserving and non-meaning-preserving) that routinely occur in the specification development process, and codifying these steps in an evolution transformation library. An analyst creates or modifies a specification by invoking a transformation from this library. ARIES assists by focusing the analyst's attention on a family of transformations which are appropriate for achieving specific effects while working with a specific presentation type.

Chapter 2

An Example of Use

The following scenario illustrates some of the principal capabilities of ARIES, and shows how they can be employed during requirements analysis. The scenario illustrates what we believe is a typical episode during the development of requirements specifications.

The scenario consists of the following steps:

1. The analyst reviews a partial requirements description, developed via previous interaction with ARIES;
2. A deficiency in the current requirements description is identified;
3. The analyst decides how the requirements must be modified in order to correct the deficiency;
4. An evolution transformation is selected which effects the modification;
5. The modified requirements are reviewed, setting the stage for the process to repeat.

It shows how presentation and evolution tools are combined to support the requirements analysis process. Other capabilities of ARIES, such as reasoning support, will be explored later in the report.

The context of this example is a requirements specification for the Federal Aviation Administration's Advanced Automation System [41], discussed in Section 1.6.2. We will focus on one functional area, the process of transferring control of aircraft between controllers and facilities, known as "handoff." Requirements pertaining to handoff are recorded in several folders, the most important being a folder named handoff. At the beginning of the scenario, the requirements for handoff have been partially identified and formalized; the

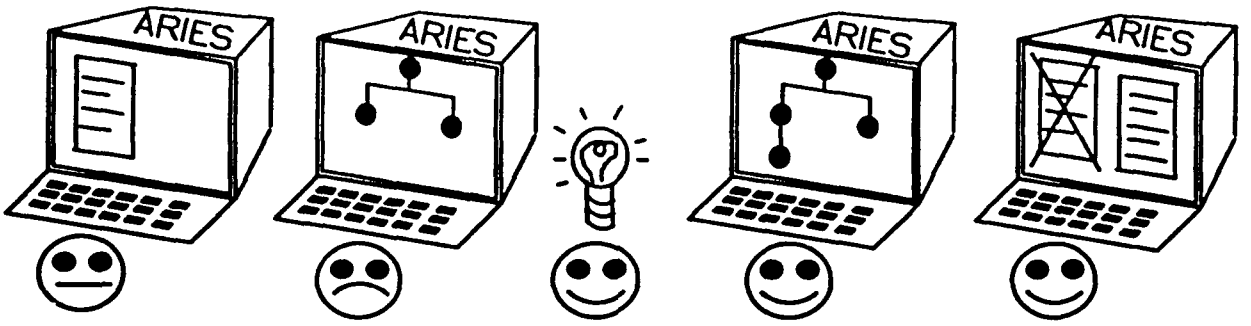

Figure 2.1: Review and modification of requirements

analyst needs to review the requirements in this and other folders, in order to identify omissions and inaccuracies.

2.1 The analyst's view of ARIES

When an analyst uses ARIES, he or she interacts through a set of windows, via the X window system. Figure 2.2 shows an ARIES interface in use. The main window, called the Manager Window, is shown at bottom. It is used to control the overall status of the ARIES session, with functions to load folders into the knowledge base, to select which folders to open in the ARIES knowledge base, and to control how to search the knowledge base. It is also used to construct presentations of folders and of knowledge base objects. Each presentation has its own window. Thus interaction with ARIES typically is conducted through multiple windows at once. The Manager Window provides a common documentation facility for all presentations: as the mouse is moved over items in the presentation windows, such as depictions of specification components or function buttons, descriptions of the items appear in the Manager Window.

At the start of this scenario, the analyst opens the handoff folder, and requests a presentation of it. He or she then proceeds to examine the contents of the folder, inspecting individual components, and possibly requesting analysis functions on components, looking for errors.

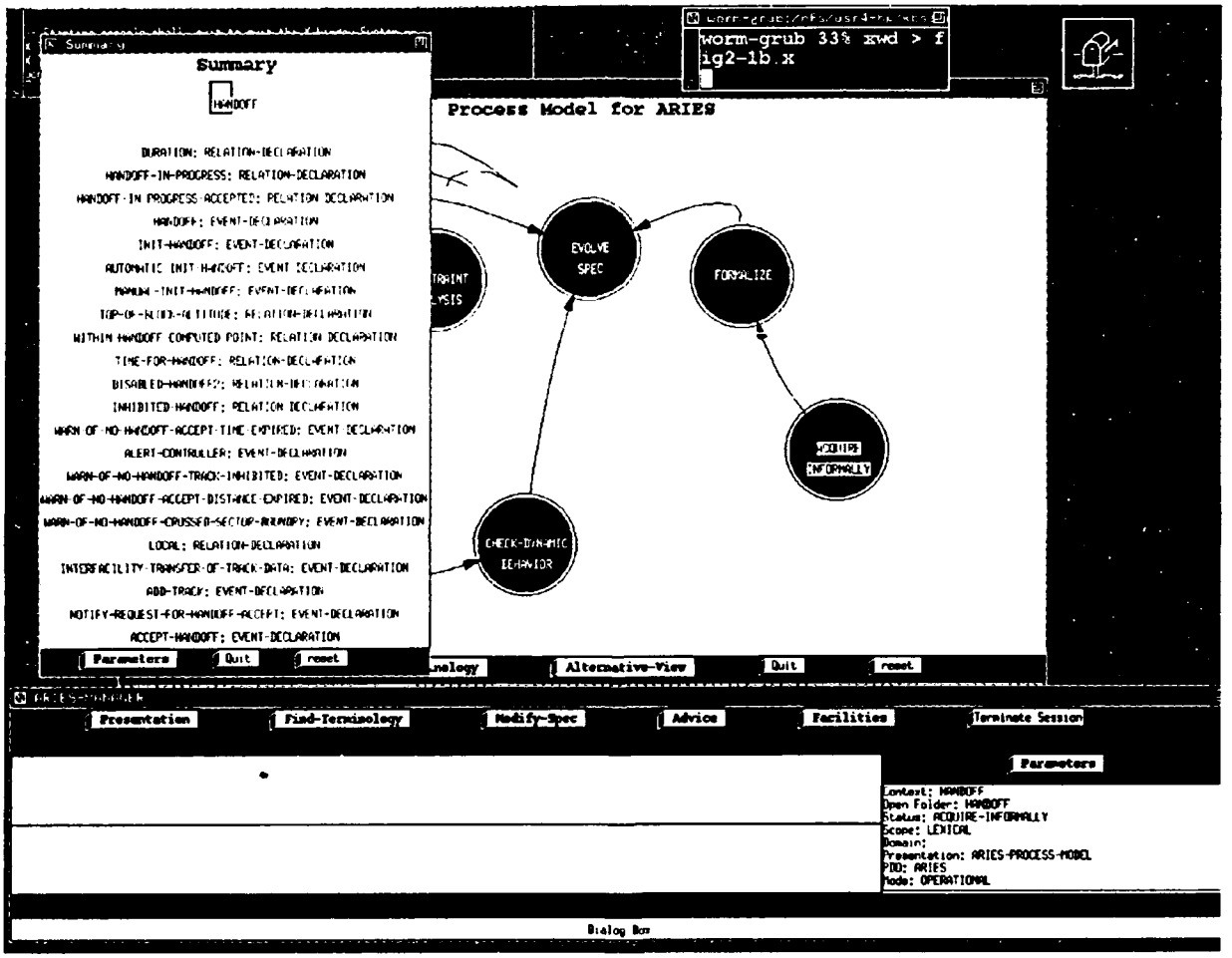

Figure 2.2: Overall view of the ARIES interface

2.2 Reviewing requirements

The analyst can make use of a variety of presentations in viewing the contents of the handoff folder. The figures in this chapter show some of presentations that can be employed. Figure 2.3 shows an overview of the handoff folder. The contents of the folder consists of declarations of *events*, i.e., actions which may be performed by the system or by external agents, and *relations* between objects. Other relevant information, such as the types of objects that can participate in the relations and events, happen not to be defined in this folder, but are instead defined in other folders to which this folder makes reference.

Graphical presentations are useful for showing the relationships between definitions. Figure 2.4 shows one such presentation, the event taxonomy for the event init-handoff. Event taxonomy diagrams show the relationships between generic and specialized event descriptions. It has the property that functional requirements are inherited from the generic descriptions to the specialized descriptions. This diagram shows the relationship between three events:

- init-handoff, which initiates the process of handing off control,
- automatic-init-handoff, which is performed by the air traffic control system to initiate handoff automatically, e.g., when a aircraft approaches an airspace boundary, and
- manual-init-handoff, which describes handoffs initiated by controller command.

The definitions of these events are all contained in the handoff folder. Each bubble in the diagram has a two-line label; the top line is the name of the definition, and the bottom line is the name of the folder that contains the definition. The names are truncated in the diagram; moving the mouse over the bubble causes the system to display the full names.

In order to see what requirements are associated with init-handoff, its definition must be viewed in detail. In this case the medium used is natural language. Natural language generation makes it possible to compare formal descriptions directly against informal requirements drawn from natural language documents or acquired from clients, making validation much easier. Figure 2.5 shows the English presentation of init-handoff.

2.3 Performing a modification

The description of init-handoff in Figure 2.5 omits an important detail: it does not allow for the possibility that handoff has been disabled. The analyst determines that a feature must be added so that handoff of individual aircraft can be enabled or disabled.

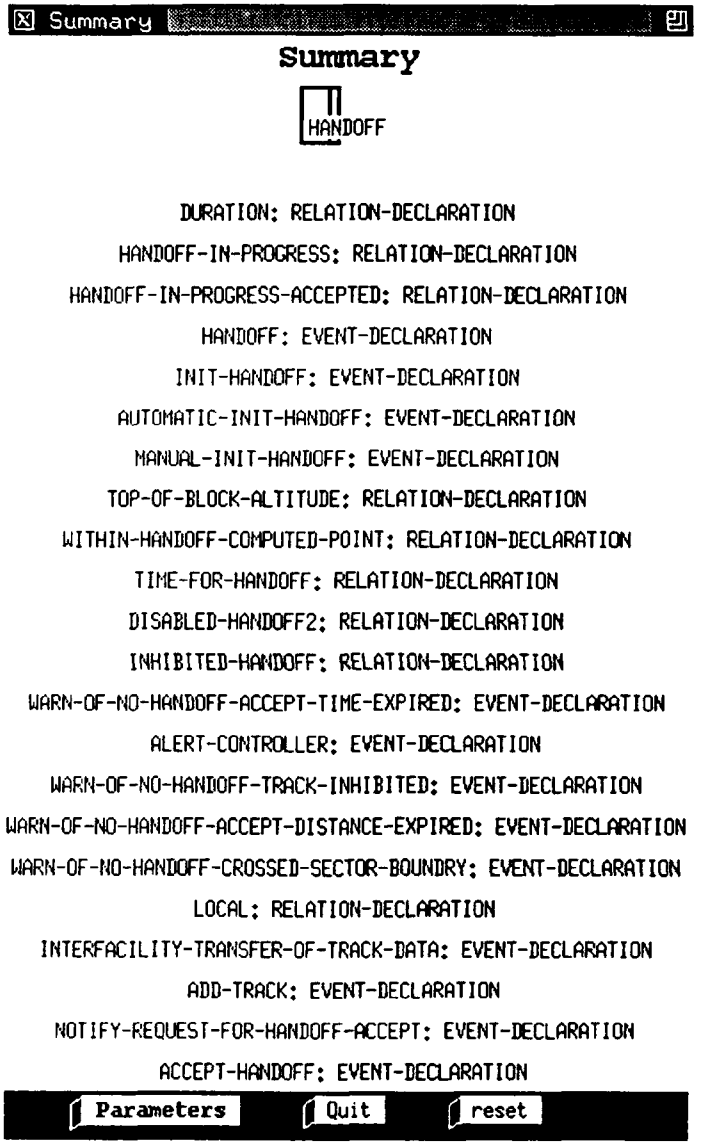

Figure 2.3: Overview of the handoff folder

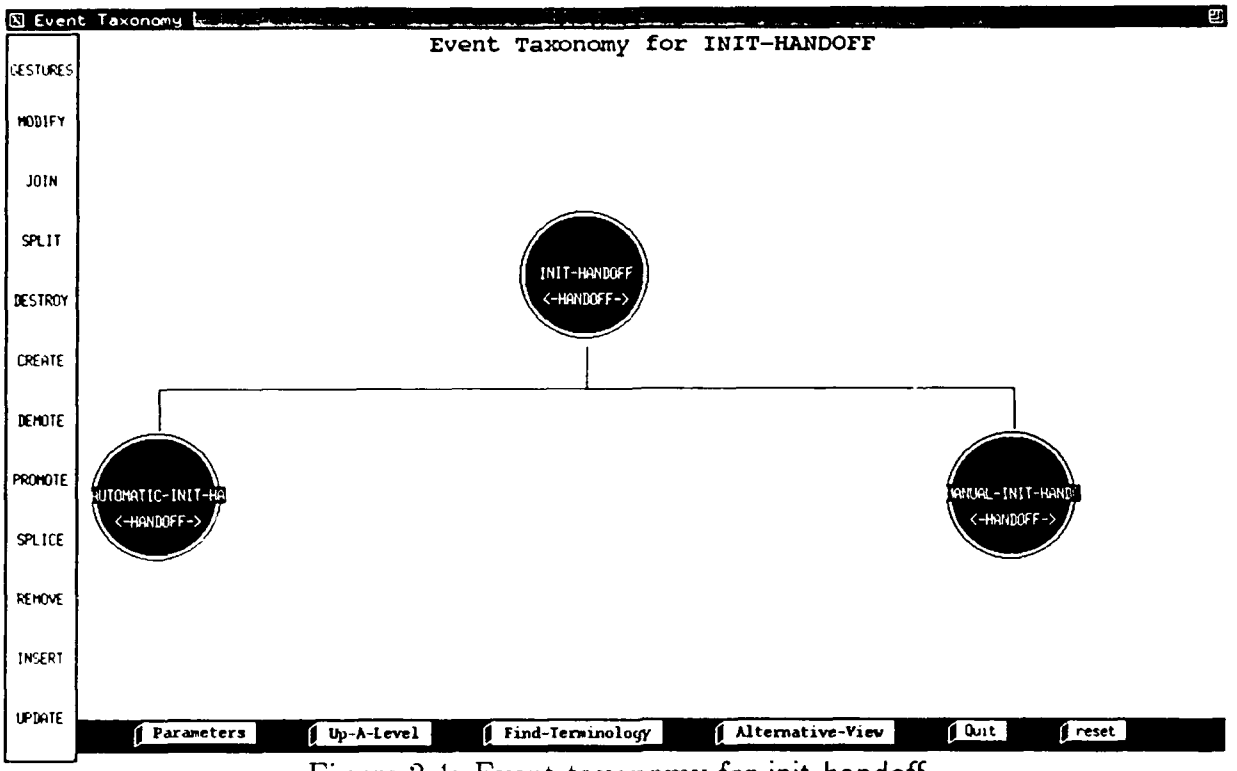

Figure 2.4: Event taxonomy for init-handoff

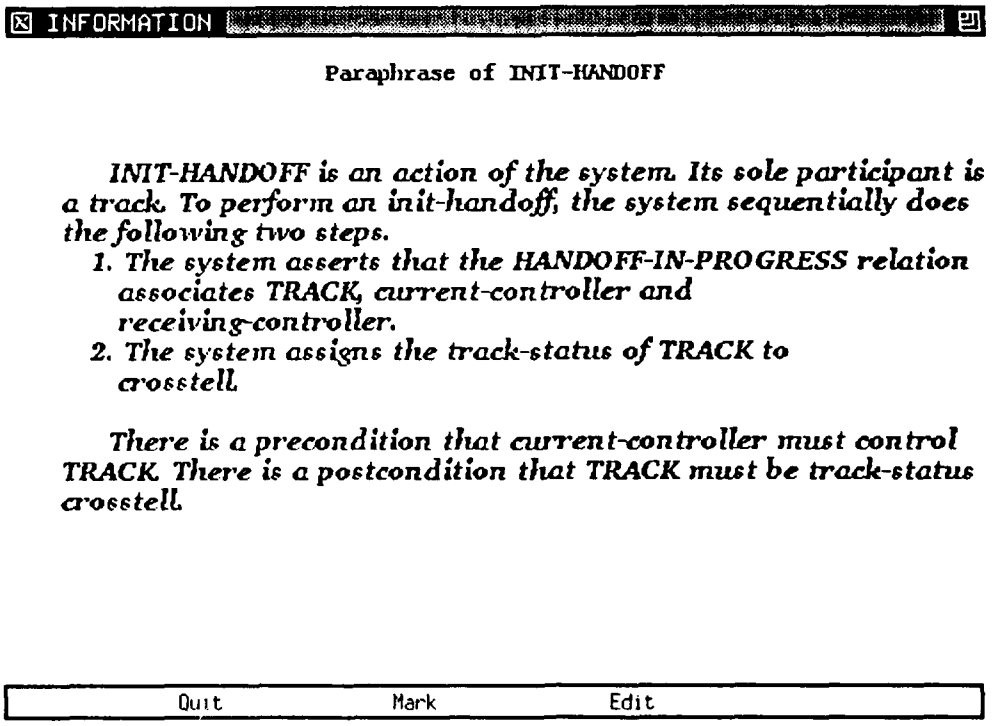

Figure 2.5: English paraphrase of the init-handoff event

This change is an instance of a general class of modification steps: whenever an event operates on a class of objects, it might be useful to add an enabling condition which must be satisfied in order for the event to be initiated. Evolution transformations were developed in order to carry out such stereotypic modifications. The transformation that applies this this case is called define-and-check-enabling-state. This transformation performs the following modifications:

- it defines two states, an enabling state and a disabling state, for some class of object;
- it defines two new events, one which moves an object from the enabling state into the disabling state, and one which moves an object from the disabling state into the enabling state;
- it adds a new precondition to an event selected by the user, ensuring that the event will not be initiated if it operates on an object in the disabling state;
- if the event has specializations, the precondition is propagated into the definitions of each specialization, ensuring that the specialization relationships are preserved.

The analyst may retrieve the appropriate evolution transformation by selecting one of the generic gestures shown on the left side of the window in Figure 2.4. These gestures are applicable to any presentation that is rendered graphically; the specific meaning depends upon the types of objects and relationships shown in the presentation. In this case, section of the gesture "modify" causes ARIES to retrieve and present in a menu all the transformations which have the effect of modifying event declarations (because event declarations are the content of this particular presentation). The transformations themselves are represented within the ARIES knowledge base, and hence the analyst can use ARIES to study the details of particular transformations in order to determine the appropriate one to apply. Once the analyst directs ARIES to apply a particular transformation, further interaction occurs between analyst and system to provide (or override default values for) the transformation's input parameters.

Once the analyst has selected the transformation, he or she enters into a menu the input parameters for the transformation. The system attempts to be helpful by filling in default values; for example, since the event taxonomy was presenting the init-handoff event, that event is suggested by default as the event that will be modified. Figure 2.6 shows the parameter menu just before the transformation is applied. The analyst has supplied names for the new states that are created: enabled-handoff and disabled-handoff. Once the analyst is satisfied with the inputs, he or she clicks on the "Done" button on the menu, and the transformation is applied.

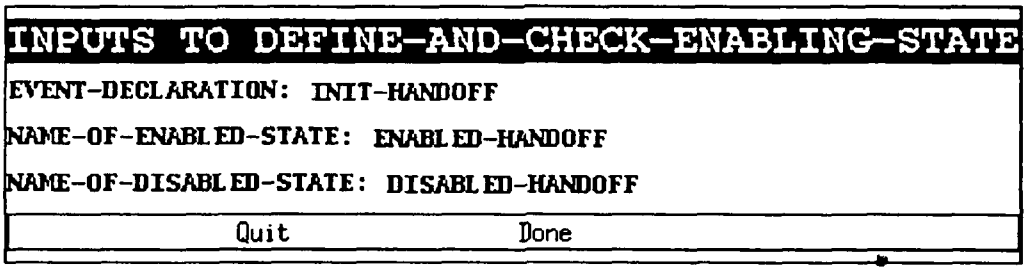

Figure 2.6: Parameter menu for transformation

2.4 Reviewing the results of the transformation

The effects of the transformation are not immediately apparent, since the analyst initiated the transformation from the event taxonomy presentation, and the transformation preserves the event specialization relationships depicted in that presentation. One way of viewing the change is to generate a new English paraphrase and compare the old version against the new version. Figure 2.7 shows the new paraphrase of init-handoff. Note that the precondition of the event has changed: it now includes the requirement that disabled-handoff not be true of the track.

Figure 2.8 shows a paraphrase of manual-init-handoff after the transformation has been applied. One can see from the figure that the same precondition has also been added to the requirements for manual-init-handoff, because it is a specialization of init-handoff. If one were to paraphrase automatic-init-handoff one would find a similar modification there. The reader may also note that the paraphrase so far says very little about what manual-init-handoff actually does—one would expect subsequent modifications to elaborate this definition.

Finally, the analyst might wish to view the new states and transitions that were created. Figure 2.9 is a state transition diagram containing the new states and transitions. Two new states have been defined, named enabled-handoff and disabled-handoff. In addition, two transitions were defined, one which transitions from the enabled state to the disabled state, and one which transitions from the disabled state to the enabled state.

State transitions are simply a subclass of event in the ARIES model. Consequently, any presentation that is suitable for viewing event definitions can also be used for viewing state transitions. Figure 2.10 shows a Reusable Gist presentation of enter-enabled-handoff. [1] In this view, enabled-handoff and disabled-handoff appear as unary relations on tracks. enter-

[1]The screen image refers to this language as "Refinable Gist." The name of the language has since been changed to Reusable Gist.

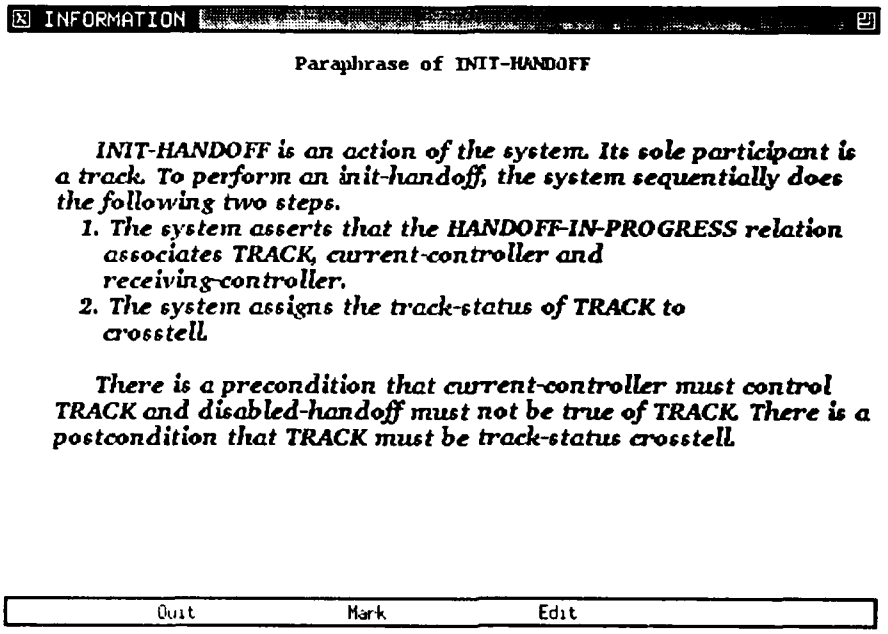

Figure 2.7: New English paraphrase of init-handoff

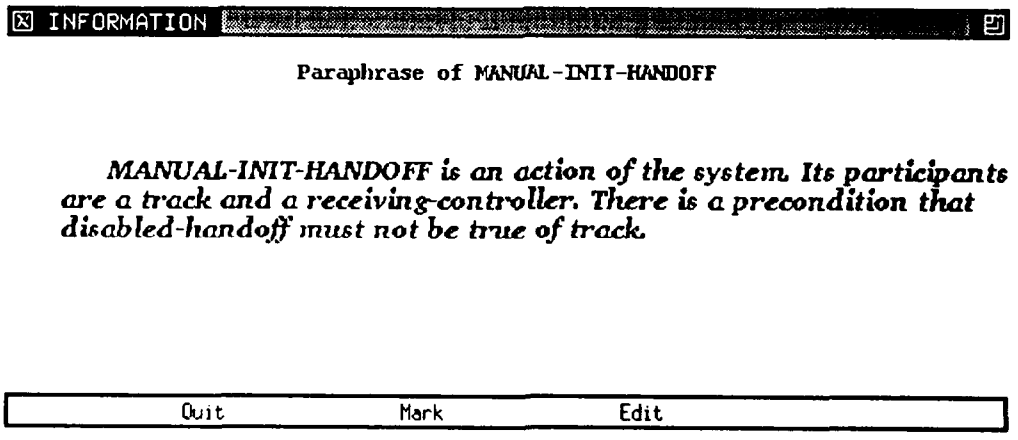

Figure 2.8: English paraphrase of modified manual-init-handoff

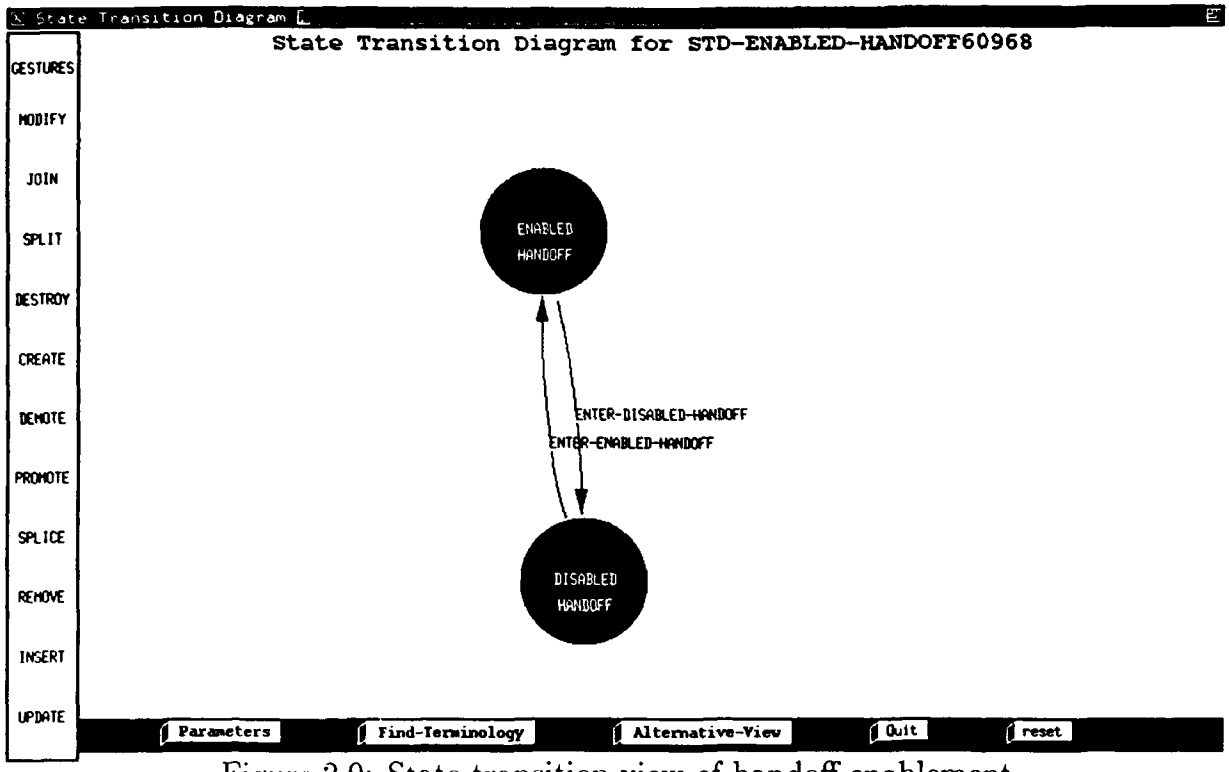

Figure 2.9: State-transition view of handoff enablement

enabled-handoff's definition is no more and no less than that implied by the modification: it asserts that disabled-handoff becomes false and enabled-handoff becomes true. If this event has special requirements in its own right, they will have to be added by the analyst.

2.5 Summary

This scenario illustrates some important capabilities of the ARIES system:

- Specification fragments may be viewed via a number of different presentations. This helps analysts to discover gaps and errors in requirements descriptions.
- Analysts can readily switch between formal views of specifications (e.g., Reusable Gist), informal views (e.g., English), and graphical views (e.g., state transition diagrams).
- Requirements specifications in ARIES typically make extensive use of specialization hierarchies. This is one of the mechanisms which ARIES provides to support requirements reuse, and to help ensure requirements consistency.

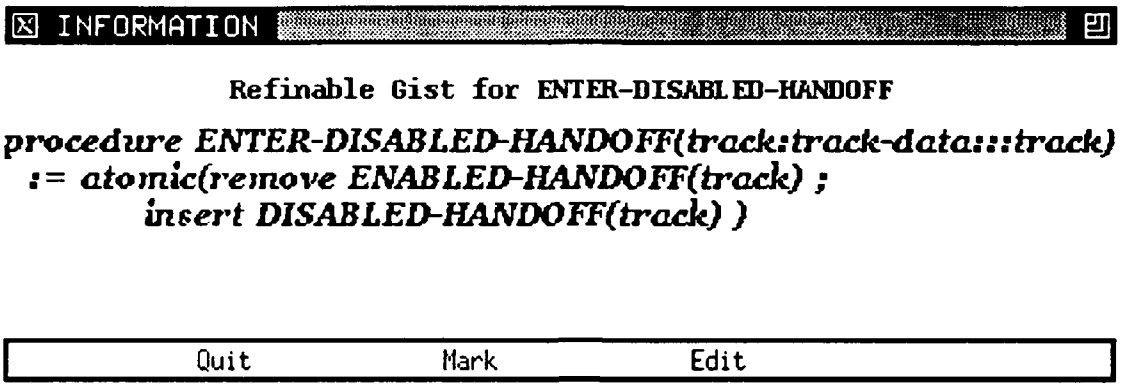

Figure 2.10: Reusable Gist view of enter-enabled-handoff

- Evolution transformations facilitate the process of modifying and elaborating requirements specifications. A single transformation can perform a number of modifications, while maintaining consistency between specification components.

Chapter 3

Representation Issues

The chapter is concerned with how knowledge pertaining to requirements engineering is represented in ARIES. The goals for supporting specification evolution impose severe challenges for the design of representation and reasoning capabilities. The following are the most important challenges.

- An extreme breadth of knowledge must be expressible. Specifications of system behavior, definitions of domain terminology, system organization, and nonfunctional characteristics must all be representable. The system must even represent significant information about itself. The transformations are one example of such an introspective representation. Another is the system's own model of different kinds of requirements objects and their relationships, called the ARIES Metamodel.
- Both strong and weak expressivity are needed. Strongly expressive constructs are needed in order to define concepts precisely, especially those that will appear in detailed specifications of behavior and system invariants. Examples of such constructs are temporal and higher-order logical operators. At the same time, analysts use less expressive, but more convenient, notations during initial acquisition of requirements. Examples include state transition abstractions and data flow abstractions.
- Partial sharing between representations must be supported. ARIES's design supports multiple projects and analysts simultaneously. It should be possible for one person to work on aircraft handoff, another to work on flight plan processing, and another to work on an entirely different requirements specification. It is not feasible for everybody to operate on a single monolithic knowledge base. The folder structuring mechanisms were developed in order to meet this need.
- Automated reasoning capabilities are required, in spite of the high expressivity of the representation. It is generally recognized that highly expressive languages are harder

to reason about, both for machines and for people. We have been deeply concerned with how to provide sophisticated automated processing without compromising on expressiveness.

- The internal knowledge representation must not be too closely tied to any one acquisition medium. This makes it possible to present information using a variety of notations. The internal representation must be able to support a variety of presentations, and in fact it should be possible to define new presentations more or less at will.

3.1 Basic underlying semantics

The basic units of system descriptions in ARIES are *types*, *instances*, *relations*, *events*, and *invariants*. The types, instances, and relations are needed to represent the entity-relationship models common in requirements engineering (e.g., [34], [10]). The representation does not have any of the limitations common in models derived from programming languages or relational data models:

- Each type can have multiple subtypes and supertypes. This contrasts with languages such as Refine in which each type can have at most one supertype. Assuming at most a single supertype makes certain kinds of processing such as inheritance easier to compute. However, it limits the ability to express entity-relationship models of domains.
- Each instance can be an instance of any number of types simultaneously. Again, this freedom is needed in order to represent complex entity-relationship models.
- Relations hold among any types of objects. This contrasts with relational data models, in which relations are defined over "atomic domains"—integers, strings, etc. It also contrasts with frame-based systems in which relations are stored only as slots on some special class of object called a "frame" or a "knowledge base object."
- Relations need not be binary, but can have arbitrary arity. This makes it unnecessary to encode ternary relations as binary relations on some artificial object.
- Relations are fully associative. This contrasts with approaches in which relations must be defined in pairs, one mapping from the domain to the range and one mapping from the range to the domain.

Events subsume actions of external agents, such as the move actions shown in Figure 3.1, as well as system processes, such as the handoff initiation actions in Figure 2.4. Events have

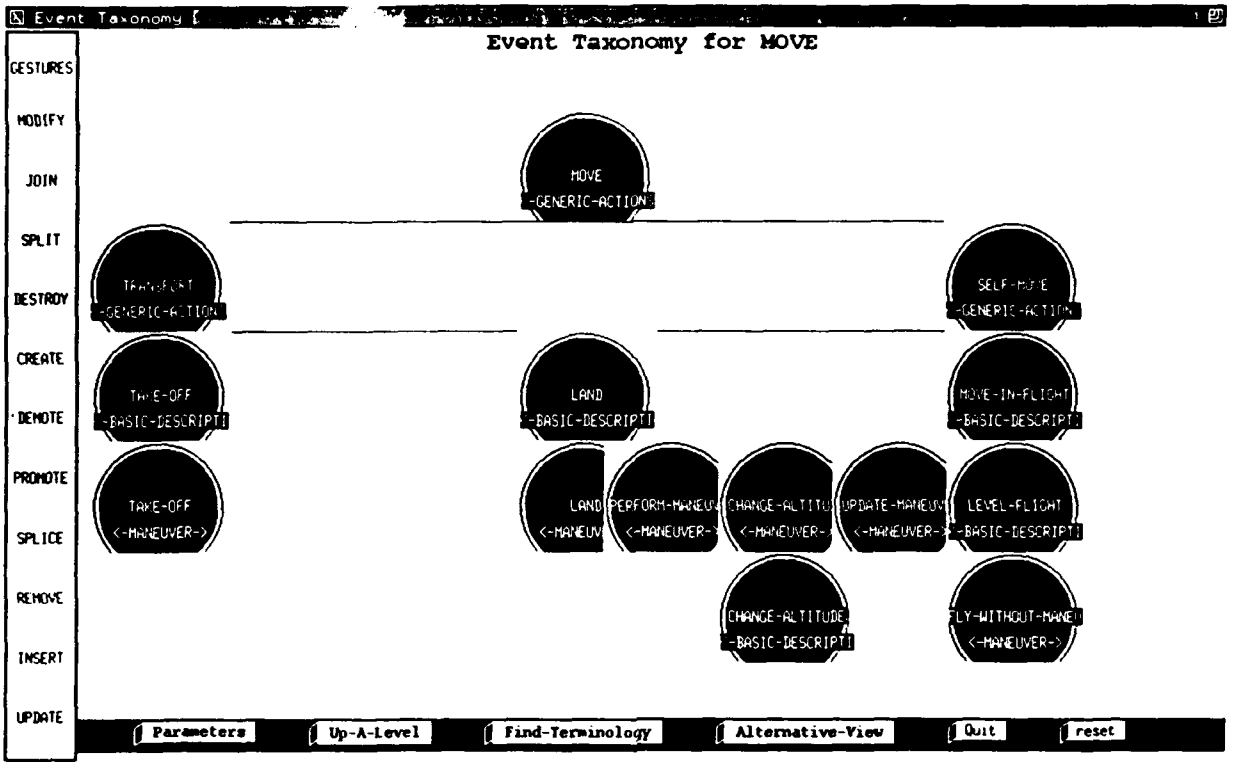

Figure 3.1: Taxonomy of kinds of motion

duration, possibly spanning multiple states in succession, and involving multiple entities of the system. This contrasts with frameworks such as ERAE, in which events are assumed always to be executed instantaneously.

Events can have preconditions, postconditions, and methods consisting of procedural steps. They may be explicitly activated by other events, or may occur spontaneously when their preconditions are met. They may have inputs and outputs. However, event definitions can affect the state of the system in ways other than generating outputs: they can assert and remove relations between objects, and create and destroy objects. A variety of behavior models can be mapped onto this event model: for example, state transitions in state transition diagrams are modeled internally as events in ARIES.

Invariants are predicates which are required to hold, either at all times or whenever a particular event is active. Many requirements on system behavior are naturally expressed as invariants, as are nonfunctional requirements. Figure 3.2 shows an example of an invariant that holds in the air traffic control domain between the scan period of a radar, the number of safe contacts for an air traffic control system, and the time before aircraft reach an air sector boundary.

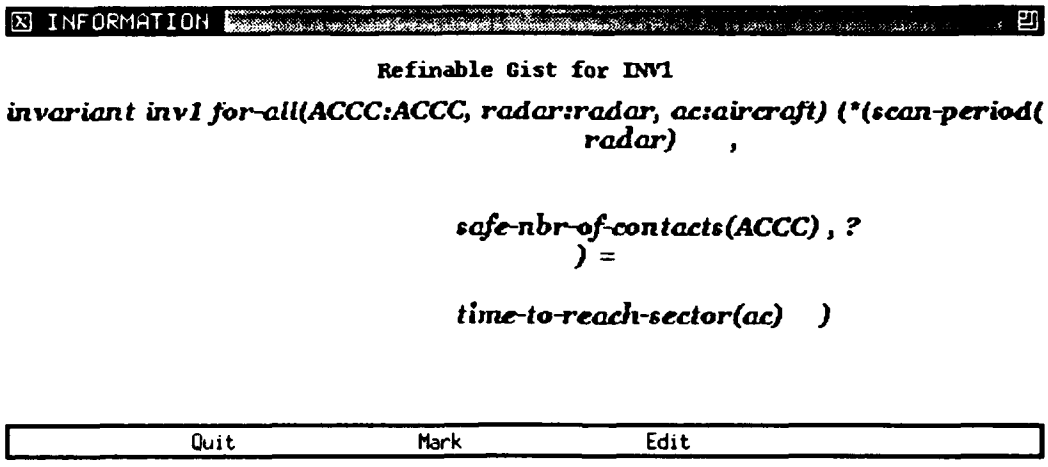

Figure 3.2: An RG presentation of an invariant

3.2 Advanced semantic features

3.2.1 Specialization in ARIES

In comparison to many common knowledge representation systems, ARIES makes more extensive use of specialization hierarchies; it supports specialization hierarchies for relations and events as well as types. Several important technical concerns had to be addressed in order to ensure that such specialization hierarchies are meaningful.

Specialization hierarchies establish subsumption relations between terms. That is, given two concepts S and T, S is a specialization of T if the following is true:

$$\forall(x)S(x) \Rightarrow T(x).$$

Specialization hierarchies of relations and events are defined in a similar manner. If we have two binary relations $R(x,y)$ and $S(x,y)$, R is a specialization of S if

$$\forall(x,y)R(x,y) \Rightarrow S(x,y).$$

Unfortunately, the above definition of specialization only makes sense when the specialization and the generalization have the same number of parameters, and parameters correspond. With multiple-arity relations and events, developed at different levels of abstraction, this is rarely the case.

Consider, for example, two events, takeoff and move. Intuitively, it would make sense for takeoff to be a specialization of move: if an aircraft is taking off, it is also moving.

```
event takeoff[ac : aircraft]
    outputs (destination := ac.location-of)
    roles (actee := ac, actor := ac)
    specialization-of move
    precondition ac.location-of is-a ground-location
    postcondition ac.location-of is-a air-location ∧ in-flight(ac)

event move [actor:agent, actee:physical-object, destination:location]
    precondition ¬ location-of(actee,destination)
    postcondition location-of(actee,destination)
```

Figure 3.3: Definitions of takeoff and move

However, the two events are likely to have different parameters. In the ARIES model, takeoff takes as input one parameter, the aircraft taking off. move, on the other hand, has three parameters: the object being moved, the agent doing the moving, and the location that the object will be moved to. A simple logical implication between the two concepts cannot be drawn, because the parameters of the two concepts do not match up. Note that most schemes for reuse of process descriptions, such as that of Lubars [54], classify processes and events in terms of their inputs and output. They would thus be unable to establish a relationship between these two concepts.

The solution to this problem that we provide in ARIES is to *reify* the events and relations, i.e., to treat instances of them as objects. When an event starts, an object representing the event is created; when the event completes, the object is destroyed. Parameters of the events become attributes of the corresponding event objects. The parameters are not identified by whether they are inputs or outputs to the process or relation, but rather according to the semantic role they play. We use as a starting point the roles defined in the PENMAN Upper Model for modeling natural language, such as actor, object, beneficiary, location, etc. It is these roles that become the attributes of the associated event objects, and serve as the basis for classification. Other roles may be defined as specializations of these domain-independent roles. This approach results in effective classification because in ensures that concepts classify in a manner similar what people would expect given the meaning of the natural-language concepts used to describe a particular event or relation.

In the case of takeoff and move, subsumption is defined as follows. Figure 3.3 shows definitions of takeoff and move rendered in Reusable Gist, our formal specification language.

Move actions are represented as objects with three attributes: actor, actee, and destination. These are the names of the input parameters in the declaration of move. Takeoff actions are modeled as having four parameters and roles. One of them, ac, is an input parameter—the aircraft taking off. Another, destination, is an output parameter, bound to the aircraft's new location when the takeoff is completed. Two other roles, actee and actor, are bound to the value of the input parameter ac. (Note that these bindings are specified in the roles field of takeoff, used to indicate semantically meaningful roles that are not explicitly represented as input data or output data.) When a takeoff event is initiated, a corresponding object is created, with attributes ac, actor, actee, and destination, corresponding to the parameters and roles. By the semantics of term subsumption, making takeoff a specialization of move means that every event object describing a takeoff must also be a well-formed move object.

Note that when an event is a specialization of another event, it must inherit the preconditions and postconditions of the events that it is a specialization of. Thus takeoff inherits the precondition and postcondition of move, meaning that the location of the aircraft at the end of the takeoff is different from that at the beginning of the takeoff. Importantly, this serves as an additional consistency check, something which is useful when we define methods for some of these events. Suppose that an analyst describes the procedure for takeoff, which includes receiving clearance from the controller. If the clearance is denied, the takeoff is aborted. But now the declared specialization hierarchy is inconsistent with the conditions on move: it is possible to "execute" the takeoff procedure and have no move occur. The creation, detection, and resolution of such inconsistency is central to the requirement development process. In this case, to remove the inconsistency, an analyst could define within a local folder a specialization of takeoff called successful-takeoff, and make successful-takeoff the specialization of move instead of takeoff.

3.2.2 Parameterized concepts

The approach to specialization described above relies upon all concepts having multiple roles, each of which have values assigned to them. Roles may be defined in terms of other roles, e.g., the actor role of takeoff is defined in Figure 3.3 to be the same as the input of takeoff. It is also possible to define concepts in which roles are declared but unbound. Such concepts are called *parameterized concepts.*

An example of the use of parameterized concepts is the following. The ARIES knowledge base contains a reusable definition of the concept of "tracker," i.e., a system that tracks the movements of objects such as aircraft. Any particular tracker has a desired accuracy range, such that the true location and predicted location of objects are within that accuracy range. In the ARIES representation, the accuracy value is represented as a role, which must be bound for any particular tracker. The type of object being tracked is also a role which must be bound to a particular domain type, e.g., commercial-aircraft or missile. When a

particular tracker is instantiated, these roles must be bound.

3.2.3 Higher-order operators

Higher-order operators were included in the ARIES Metamodel in order define properties that hold for classes of concepts. It is often the case that an application-specific concept must satisfy some generic property, expressed in higher-order terms; referring to the higher-order property can help to ensure the accuracy of the application-specific definition.

A simple example of a concept that is defined over a class of concepts is the concept of a symmetric relation. The concept symmetric is defined as follows:

```
implicit relation symmetric(r:relation-declaration)
   iff
      arity(r, 2) ∧∀ (a:entity, b:entity) r(a,b) ⇔ r(b,a)
```

This defines symmetric as a unary relation that holds for other relations. A relation r is symmetric if its arity is 2, and if for all objects a and b, if $r(a, b)$ is true then $r(b, a)$ is true.

3.2.4 Temporal operators

Temporal operators make it possible to express invariants that hold between system states at different points in time. For example, the temporal predicate P as-of Q is true if P is true at some point in time in which Q is also true. The temporal predicate start P is true if P is true at the present time, and was not true immediately previously. The temporal operators in the ARIES representation are the same as those defined in Reusable Gist.

3.3 Scope of knowledge represented

This general framework for representing knowledge is used in ARIES to represent all domain knowledge in the system, as well as substantial parts of the ARIES system itself.

System descriptions are themselves stored in ARIES as a knowledge base of types and relations. The knowledge base not only contains definitions of domain types such as aircraft, it also contains definitions of types of knowledge base objects, such as type-declaration. In addition to relations between domain types, such as the altitude relation between aircraft and numbers, the ARIES knowledge base includes relations between knowledge base objects,

such as the generalization relation between concept declarations and their supertypes or generalizations.

Transformations in the ARIES system are represented internally as events. This makes it possible to define declarative specifications of transformations, including their inputs, outputs, preconditions, and postconditions.

This self-representing approach affords several advantages. Having declarative specifications of ARIES components makes it easier for engineers to understand what those components are and what properties they have. Tools for describing and explaining knowledge base components, such as the the ARIES Paraphraser for generating natural language, may be applied to components of the ARIES system itself.

At the same time, this self-specification approach results in a certain amount of duplication. The ARIES representation is a representation for specifications, not for implementations. Thus for each specified construct there is a corresponding implementation of that construct. For example, for each transformation in the system there is both a declarative specification, represented using the ARIES representation, and an implemented Lisp function that is called in order to execute the transformation. The correspondence between specification and implementation is automatically maintained. At one point during the development of ARIES the implementations were generated automatically by the ARIES system using a compiler from the ARIES Metamodel into Lisp. This approach was abandoned simply because the compiler was constantly being developed and modified, making it less readily usable. Instead, transformations are defined using special Lisp macros that expand into both partial specifications and Lisp implementations of the transformations. In the more rugged ARIES systems planned for the future, such a self-specifying and self-compiling approach offers distinct advantages.

3.4 Features not supported

This framework might appear to include just about every feature common to knowledge representation frameworks. However, there are some features that have been intentionally omitted from the framework, mainly because existing features provide most of what of these omitted features provide. The most significant such feature is the ability to overload names. Some knowledge representations allow one to give two different relations or events the same name, as long as they apply to distinct classes of objects. ARIES places an emphasis on defining hierarchies of concepts. If a concept applies to two different categories of objects, the preferred approach is to determine what the shared meaning is, to define the shared meaning as a single concept, and to define the individual concepts as specializations of the shared concept. For example, instead of defining separate methods for each class of

object in an object-oriented programming system, one first specifies the general properties of the methods as an event. One then defines methods for individual classes of objects as specializations of the generic event. This helps to avoid the problem of methods that have the same name but incompatible functionality.

In a full KBSA system some features not currently supported will have to be provided. In particular, some sort of overloading capability will be needed, even though specialization hierarchies will continue to be used. Since it is currently possible to use the same name for two concepts as long as the two definitions appear in different folders, an overloading feature was not given high priority in ARIES.

3.5 Mapping narrow formalisms onto the underlying representation

In order for a representation such as the ARIES Metamodel to be workable, it is necessary to be able to map the underlying representation onto the notations which analysts wish to view and manipulate. Those notations are narrower in scope than the underlying representation, which incorporates much more information than analysts will be concerned with at any one time.

3.5.1 Mapping via abstracted representations

For most notations, including diagrammatic notations such as type taxonomies, the mapping from representation to notation proceeds in two stages. The first stage is a mapping from the detailed underlying representation to a narrower abstracted representation. The second stage maps the abstracted representation onto the notation that the user actually sees.

The taxonomy diagrams in ARIES are examples of this two-stage approach. The abstracted representation in a type taxonomy diagram consists of a specific class of objects, namely type declarations. Three relations are depicted in the presentation: the generalization relation associating concepts with their generalizations, the name relation associating concepts with their names, and the component relation associating concepts with the folder in which they are defined. This abstraction of the full knowledge base is then presented in the display in the form of a tree: each type declaration appears as a node in the tree, the generalization relations appears as edges in the tree, and the other relations are used to generate labels for the nodes in the tree. The process of associating abstracted representations with presentation styles is discussed in detail in Chapter 4.

Non-orthogonal presentations are supported in the following manner in this system. As indicated earlier, system descriptions are represented as typed objects associated via relations, just as ARIES itself models knowledge using types and relations. This information is modeled in a language called AP5, which provides relational data representations as an extension to Lisp. Some of the types and relations in ARIES are defined as primitive relations. Others are defined logically as predicates over other relations. If a relation $R(x, y)$ is defined in terms of some other predicate P, $R(x, y)$ is true for every x and y such that $P(x, y)$ is true. Other types and relations are defined procedurally via Lisp functions. If a relation $R(x, y)$ is defined using a Lisp function f, $R(x, y)$ holds whenever the Lisp expression (f x y) evaluates to a non-nil value. Examples of relations that are defined procedurally are information flow relations. Information flow generalizes data flow: it subsumes any access by a process to properties of objects in its external environment. The information flow relations in ARIES are defined via procedures that examine the preconditions, postconditions, and methods of events to determine what information accesses and modifications are performed.

3.5.1.1 Updating non-primitive relations

For non-primitive relations, an important technical issue is how to update them. For example, when a relation $R(x, y)$ is defined in terms of a predicate P, and a component of ARIES asserts that R is true for some x and y, some other knowledge base updates must be performed so that $P(x, y)$ is also true. It is not always possible to determine from the definition of P what knowledge base updates must be performed. This problem is addressed in one of three ways.

One approach is to make use of AP5's add method and delete method features. AP5 allows a programmer to specify a complex knowledge base update to perform, or a function to execute, whenever an attempt is made to add or delete a tuple from a relation.

A second approach makes use of rules that are activated automatically in response to assertions or retractions of other relations. For example, each term reference appearing in a folder relates to some concept in the knowledge base, depending upon what terms are defined in the folder and what other folders are used by the folder. As new term definitions are added, and links to other folders are added and removed, rules automatically update the reference relations. Thus if a folder contains a term-reference named level-flight, the folder does not contain a definition of level flight, and the folder does not use any folder with a definition of level flight, then no reference relation is asserted for this term—it is an undefined reference. If the analyst then updates the use list for the folder, making it use a folder which contains a definition named level-flight, a reference relation between the term reference and the definition is automatically asserted.

The third method of performing updates is via evolution transformations. When an analyst attempts to modify a presentation, the transformation system retrieves a set of transformations capable of performing the modification. This mechanism will described in detail in the evolution section of this report, Chapter 7. When a transformation is selected, the transformation performs a series of updates to the knowledge base that result in the abstraction underlying the presentation being modified.

3.5.1.2 When sufficient information is unavailable

It is not always possible to generate a presentation, given the information available in the ARIES knowledge base at any one time. That is, the mapping from the internal representation to the abstraction underlying the presentation may be ambiguous. Objects in the internal representation might map into the underlying abstraction in more than one possible way, and information that could determine which mapping to use is not yet present in the underlying knowledge base. In such cases, information must be added, either to the underlying representation or to the resulting abstraction, in order to complete the presentation.

One cause for this difficulty is that the ARIES Metamodel permits analysts to delay commitment as to what category a knowledge base object belongs to. An analyst can name a concept, e.g., flight, without committing to whether it is a type or an event or some other construct. This poses problems for notations that assume that constructs belong to specific categories.

Two methods are employed for handling such underspecified constructs, depending upon the particular presentation being employed. One method is simply to omit from the presentation those objects that are not known definitely to be viewable via a presentation. This method is used in the type taxonomy presentation: objects that are not declared to be type declarations are not viewable via this presentation. The other method is to assign the knowledge base objects by default to specific categories for the purpose of creating the presentation. For example, in the Reusable Gist presentation underspecified constructs are presented by default as instances. The choice of which method to employ depends upon whether the presentation is intended to present all components of a given folder, or only selected components of a folder.

A related problem arises when it is not possible to determine automatically how to present information in a presentation, when that information was not entered via a presentation in the first place. A case in point is the state transition diagram presentation. State transition diagrams are used to show how some object in a system, or the system itself, alternates among a finite set of states. For example, a radar track in an air traffic control system may alternate among various statuses, such as normal, dropped, coasting, etc.

In order to make effective use of a state transition diagram in ARIES, it was necessary to determine how to present arbitrary models encoded in the ARIES Metamodel as state transition diagrams. Nilary, unary, and binary relations are potentially viewable as states, under the following conditions. Nilary relations may be interpreted as states of the system; for example, a nilary relation called operational() might be used to indicate whether the system being specified is the operational state. Unary relations may indicate states in two different ways. They may be taken to indicate that the parameter of the relation is in a given state, e.g., enabled-handoff(track) was used in the scenario in Chapter 2 to represent the state of a track being enabled for handoff, as was shown in Figure 2.10. Alternatively, if the parameter may take one of a finite number of values, it may be that the values are to be interpreted as states of the system. Thus, operational-status(status) may be used to describe the operational states of the system, provided that status can take on one of a finite number of values, e.g., operational, training, maintenance, etc. Whenever a binary relation has one parameter that can take on one of a finite number of values, the parameter with a finite number of values may be taken to indicate states of the other parameter. Thus the relation track-status(track, status) may be taken to indicate states of tracks provided that status can take on one of a finite number of values.

The above rules only indicate what relations *might* be considered states; they do not indicate what relations *should* be viewed as states. Suppose, for example, that a model of an air traffic control system contains a relation controlling-position(aircraft, controller-position) associating each aircraft with the controller console (a.k.a. controller position) controlling that aircraft. There is a finite number of such controlling positions; does that mean that it is meaningful to treat controller positions as indicating states of the aircraft? Ultimately, this depends upon the point of view of the analyst. The analyst may consider the controller position to be a significant indicator of the state of the aircraft, or may view all controller positions as alike, in which case the choice of controller position is not significant at all.

Heuristics can be articulated for identifying relations that are likely to be viewed as states. However, such heuristics are fallible if the specification is incomplete. For example, if an incomplete specification makes no distinction among the capabilities of different controller positions, this may be evidence that controller positions do not signify states, or it may be evidence that the characteristics of different controller positions have not been fully spelled out.

Instead of using heuristics to guess what relations are likely to represent states, ARIES was designed with the presumption that it is up to the analyst to indicate what is a state and what is not. Analysts are given the option of indicating that particular relations signify states, in a particular state transition diagram. Relations signifying states are viewed as a subclass of the more general category of relations. Thus the problem of presenting a model as a state transition diagram becomes similar to the problems described earlier

in this section of constructing a presentation when the classification of the objects being presented is underspecified. As is the case with diagrams such as type taxonomy diagrams, relations are not taken to represent states unless an analyst explicitly indicates that they should be viewed as states.

3.5.2 Mapping via transformation

Another approach to mapping narrow notations onto the underlying representation is the use of transformations. Transformations are used to translate the internal representation into external notations, and to translate external notations into the internal representation. This approach is used to map textual formalisms, including Reusable Gist, Refine, and Loom, onto the ARIES Metamodel.

Some of these transformation processes are mediated by intermediate abstractions, as is the case for the diagrammatic representations described earlier. This is the approach used to generate Refine from the ARIES Metamodel. Refine does not support the use of arbitrary relations for constructing models; instead, models are constructed out of sets and maps. Instead of using unary relations, for example, one instead defines maps whose range is the set of boolean values. In order to generate Refine text, ARIES first transforms the ARIES Metamodel into a subset that can be more readily translated. That includes replacing each unary relation with a map onto the booleans.

3.6 Integrating textual and relational representations

The relational representation used in the ARIES Metamodel provides a uniformity that facilitates the development of intelligent tools. Each tool can operate on a subset of the relations in the knowledge base, or on abstract relations defined in terms of other relations. Each tool can thus focus on an abstract representation of interest to it.

There are some types of processing for which a relational representation is unsuited. Since semantic networks may contain cycles, traversal of the network must be done carefully to avoid infinite loops. Each relation in the metamodel has type and cardinality restrictions. This must be checked as the knowledge base is updated. This checking can greatly slow processing when massive updates to the knowledge base are being made, as the case when folders are being loaded from files on disk. Transformation processes, such as the processes of translating the ARIES Metamodel into Refine, are most easily expressed as mappings from textual patterns in one language onto textual patterns in another language. This is not possible if the internal representation is a collection of relations instead of text. Finally, saving and restoring sections of the knowledge base to disk files becomes complicated,

because it requires translating a nonlinear, cyclical representation into a linear textual form and back again, reconstructing the same cycles in the process.

The ARIES Metamodel has been implemented so that it has both a textual and a relational form. We use the relational database capabilities provided by the AP5 system [15] as the uniform means to access all forms of represented knowledge. However, a significant part of the processing done within ARIES involves grammatical objects (parse trees), for which we use the POPART language processing system; a sizable subset of the types and relations in the knowledge base are in fact realized in the form of parse trees. The key to combining these two facilities is to use POPART's parse trees as *implementations* of AP5's relations. This is possible because AP5 allows developers to select arbitrary data structures to implement relations. Examples of such data structures are trees, hash tables, and linked lists. The original purpose of this capability was to permit the separation of data structure selection and algorithm design. In ARIES, this capability is used to allow AP5 to treat parse trees produced by POPART as implementations of AP5's relations.

In order to allow AP5 to treat POPART parse trees as implementations of relations, we had to define an operation on parse trees corresponding to each of the primitive operations on relations, *asserting* a relation, *retracting* a relation, *testing* a relation, etc. Once this had been done, relations implemented as parse tree could then be used just like any other AP5 relation, and tools that operate on the representation need not be concerned with which relations were realized in POPART and which were not, unless those tools relied specifically upon POPART capabilities.

The following example illustrates how AP5 and POPART capabilities are integrated to implement the internal representation. Suppose that flight-plan is a ternary relation between an aircraft, its starting airport, and its designation airport. Then the following expression in Reusable Gist denotes an expression whose value is any aircraft whose starting airport is "Los Angeles" and whose destination airport is "New Brunswick":

```
any aircraft | flight-plan(aircraft,"Los Angeles","New Brunswick")
```

Entering this into the ARIES system and representing it using the ARIES Metamodel involves the following steps. First, the above Reusable Gist expression is parsed using POPART. A POPART-based language translator is then invoked to translate this expression in another POPART parse tree in a grammar for ARIES Metamodel objects. In this internal grammar the expression appears as follows:

```
(instance-retrieval
   :variable (variable :name aircraft :type entity :determiner any :time present)
   :predicate (query  :concept (reference :name flight-plan :class relation-declaration)
                      :actuals (reference :name aircraft :class instance-declaration)
```

(reference :name "Los Angeles" :class instance-declaration)
(reference :name "New Brunswick" :class instance-declaration)

Even this small example makes it obvious why the form of the internal representation is for machine processing, not human perusal!

What it identifies is a reference to some instance (the instance-retrieval form) that will be bound to a variable named aircraft (the variable form), and which satisfies a particular predicate involving the flight-plan relation (the query form).

Now let us look at how part of this expression, the query form, is actually represented by POPART. POPART represents parse trees as nested Lisp lists. The head of the list indicates the type of nonterminal of the expression, and the tail of the list consists of the fields of the parse tree node, i.e., its children. Empty fields are stored as NILs. The Lisp expression for the query is as follows:

```
(QUERY NIL NIL NIL NIL NIL NIL
       ((REFERENCE (SYMBOLIC AIRCRAFT)
                   (DECLARATION-CLASS-FIELD :INSTANCE-DECLARATION)
                   NIL NIL NIL...)
         (GIST-STRING "LosSpaceAngeles")
         (GIST-STRING "NewSpaceBrunswick"))
       NIL
       NIL
       NIL
       (REFERENCE (SYMBOLIC FLIGHT-PLAN)
                  (DECLARATION-CLASS-FIELD :CONCEPT-DECLARATION)
                  NIL NIL NIL...))
```

Each field in the parse tree always appears in the same position in the list representing the parse tree node. For example, the form indicating the relation being queried, e.g., flight-plan, always appears in the 12th position in the list. POPART automatically assigns the positions for the various fields.

The first step in defining the AP5 relations in the ARIES Metamodel is a tool which takes as input the POPART grammar for the internal parse trees and defines a collection of AP5 relations for storing and retrieving fields from the parse trees. One of these automatically generated relations is called concept-ref; it is a binary relation between query predicates and reference expressions identifying the relation being queried. The abstract operations on the relation are implemented as calls to the primitive POPART functions for manipulating these Lisp lists. For example, in order to assert a tuple of the concept-ref relation, a POPART

function is called to update the concept field of the query, which in turn updates the 12th position of the list.

Once these primitive AP5 relations are defined, other relations may be defined in terms of them. For example, tools rarely need to access reference expressions; they usually are instead interested in the knowledge base objects that they refer to. So a more abstract relation is (concept x y) has been defined, in terms of the following predicate:

```
(exists (z) (and (concept-ref x z) (refers-to z y)))
```

refers-to is another primitive relation between reference objects and the definitions that they name. refers-to is implemented internally via symbol tables which record where all concepts, instances, folders, etc. are defined in the knowledge base.

This definition of the concept relation is an elementary example of how more abstract AP5 relations can be defined in terms of other primitive relations. This approach is used extensively for defining abstraction layers on top of the ARIES Metamodel: an abstract relation is defined as a predicate over more primitive relations.

The features in ARIES for saving and restoring folders make use of this combined textual and relational representation in the following way. There is a list of those relations in the ARIES Metamodel that are not embodied in, or derived from, the textual representation. When a folder is saved out, two disk files are created. The first file consists of the textual representation of the folder. The second file consists of a list of AP5 assertions of those relations which are not represented in the textual form. The save facility traverses the objects in the folder, looking for objects about which relations in the list have been asserted. An assertion is written to the file for each such assertion encountered. Each reference to an object in the knowledge base is replaced with a Lisp form which looks up the desired object when the form is evaluated. The resulting file of assertions is much smaller and easier to process than a file of all the relations in the folder would be. When a folder is restored, ARIES first parses the textual represention file, to create the necessary knowledge base objects. The assertion file is then read and evaluated, in order to assert the additional relations on the knowledge base objects that were created.

Chapter 4

Presentation

In this chapter we focus on the presentations that users interact with, and show how they are defined and linked to the underlying knowledge base.

Our fundamental goal is to give the analyst expedient access to a wide variety of requirements information. To attain this goal we needed to provide a variety of presentations; furthermore, in the course of developing ARIES, we needed (and continue to need) to be able to easily experiment with, and rapidly modify, these presentations. This led us to choose wherever possible a *declarative* style of interface definition from which the corresponding interface can be *generated.* Alternatively, we could have chosen to hard-wire each interface individually. Our selection trades some loss of efficiency and restricted ability to fine-tune presentations in return for rapid construction of modularized interfaces. Presentation objects are organized into hierarchies and we take advantage of inheritance of properties and generic approaches to the creation of interface components.

4.1 Construction of presentations

We developed the ARIES presentation system in three stages. First, we established *presentation styles*—the boxes, arrows, labels, menus, tables, tree diagrams and flow diagrams. Presentation styles encode the syntactic, organizational features of user interfaces. These styles are defined independently of the threads which connect the interface to the semantic information in the knowledge base. Each style captures a particular approach to organizing and aligning data for display on the screen. Each style has a display approach (size, color, ornamentation), and aggregate objects have an organizational approach (layout of components). Second, we established recognition facilities to handle the response to navigation requests. A library of reusable navigation buttons was specified, each having a display

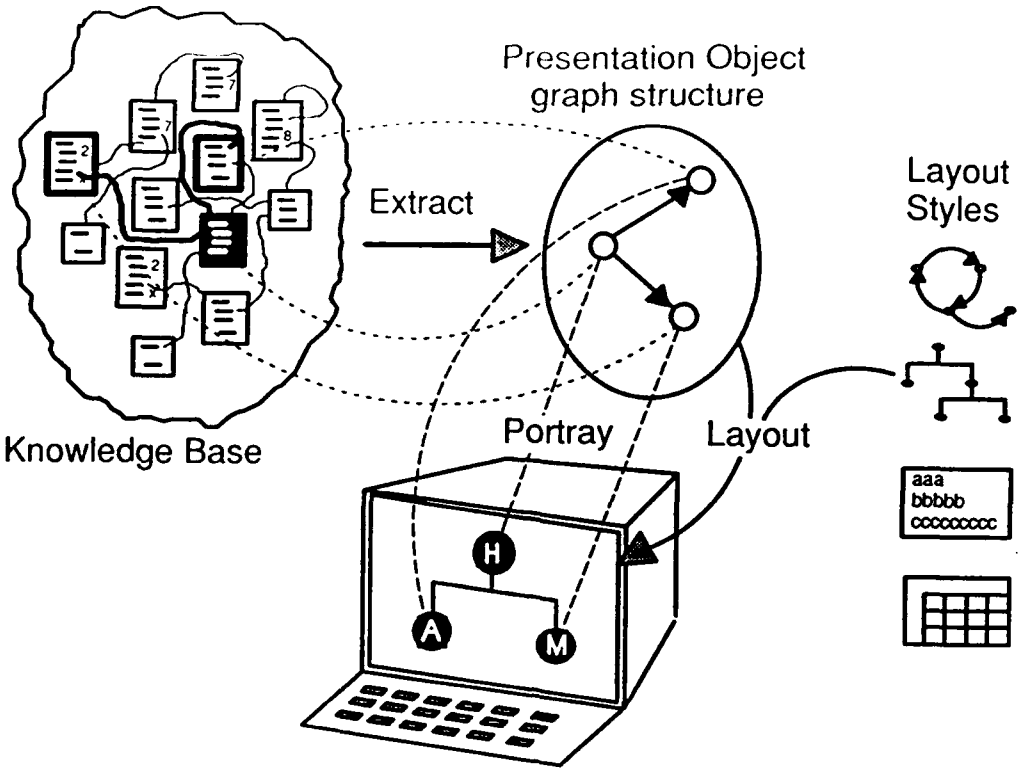

Figure 4.1: Constructing a presentation

name which appears as the label on the button, an advice text string which appears in a documentation window, and an action which occurs when the analyst selects the button. Navigation buttons can be mixed and matched for each presentation type. The third stage established the declarative presentations appropriate for the requirements/specification domain. When designing a presentation, the selection of a style does not commit to particular query for extracting a piece of the knowledge base, to the manner in which individual components within the presentation will be portrayed, nor the particular menu options that will be made available—these are further choices that the designer makes when designing the presentation itself. These presentations link user interface constructs with semantic information—items in the knowledge base. Each presentation also contains a list of editing and navigation options from the library of navigation buttons, and a style (flow diagram, tree diagram, etc).

We will illustrate how presentations are specified with two examples of presentations that are part of the ARIES system. The first example is an event-taxonomy presentation. Figure 3.1 presents the event move and its event taxonomy, that is, all its specializations. To create this display, we apply the event-taxonomy presentation to the event object move in the knowledge base. The event-taxonomy presentation finds all the specializations of the event, and displays the event together with its specializations in a tree structure.

The process of constructing a presentation involves two processes, each of which is specified

as part of the presentation definition. These processes, illustrated in Figure 4.1, are as follows.

Extracting the information to be portrayed — This consists of retrieving an abstraction of the knowledge base, as described in the previous chapter. The presentation being defined expects to be given a single object as the starting point from which to extract information from the knowledge base. In this case, the object is to be an event-declaration (one of the classes of objects in our knowledge base). The transitive closure of a binary relation finds further objects to be displayed. In this case, the relation **generalization** (linking concepts to their generalizations), in the *reverse* direction (because we want to display the *specializations* of the declaration provided, rather than its generalizations) defines the desired objects. The resulting abstract graph is a projection of the entire knowledge base, based on the generalization relation. More complex projections are achieved through the definition and use of the appropriate relations used to induce such projections.

How to portray the extracted information — unbalanced-tree is the particular style of diagram to be used to portray the event taxonomy information.

The event-declaration objects extracted as described above will be the nodes (a.k.a. vertices) of the tree, and the branches of the tree will indicate the generalization links among these nodes. Individual nodes will be displayed as shaded-box, one option for displaying the nodes of a presentation. The presentation constructor can select from a library of such "icons" differing in size, shading, geometry, and ornamentation. Each node is labeled with the name of the object it is portraying, and the folder in which it resides. Again, the presentation constructor selects the function to generate this label from a library of such labeling functions.

Thus in summary, the declarative definition of the event taxonomy presentation states the following information[1]:

- The type of the knowledge base object from which to compute information to be portrayed—event-declaration
- The binary relation to be followed to find objects to be portrayed—generalization
- The overall style of diagram—unbalanced-tree
- The appearance of nodes in diagram—shaded-box

[1]For brevity, some details have been omitted, namely directives of whether and where to include it in the various menus of possible presentations, and directives of how to respond to mouse-gestures associated with this diagram.

- The function to generate node labels—folder-and-relnames, i.e., the particular function that produces the name of the object and the name of the folder in which it resides.

As a second illustration, a state-transition diagram, as used in Figure 2.9, is an example of a more complex presentation, involving more complex layout of the arcs between nodes, and labeling of those arcs. Its declarative definition must state more information, as follows:

- The type of the knowledge base object from which to compute information to be portrayed—std, a unary relation whose values are filled in by the analyst.
- The relation to be used to find objects to be portrayed as nodes—states
- The relation to be used to find objects to be portrayed as arcs—transitions
- The function to compute the node at the start of an arc—start-state
- The function to compute the node at the end of an arc—end-state
- The overall style of diagram—flow-diagram
- The appearance of nodes in diagram—circle
- The appearance of arcs in diagram—arc-labels i.e., labeled arcs connecting to nodes on the diagram.
- The function to generate node labels—state-name
- The function to generate arc labels—transition-name

In summary, to present information in a palatable fashion, we must i) extract a chosen subset of the knowledge base information to portray, and ii) display that information in an appropriate form. The key to easy extraction of information to portray is the ability to define relations on top of the underlying knowledge representation, as discussed in Section 3.5.1. The key to displaying the extracted information is the use of appropriate presentation types—diagrams, natural language paraphrases, or formal languages.

The goals for multiple presentation acquisition and review required that we experiment with a wide variety of modes which slice up the underlying specification representation in informative ways. In order to populate ARIES with a wide variety of flexible presentations, we use a declarative style of definition, in which the system builder selects from a number of provided building blocks to compose the features of the presentation he/she desires.

4.2 Presentation implementation

The ARIES user interface builds on the Common Lisp Object System (CLOS), Common Lisp X (CLX), and the Common Lisp User Interface Environment (CLUE). With CLX, ARIES runs on host machines supporting the popular X protocol. Also, to a limited extent we have used X support to investigate multi-user issues involved in multi-display access to a single underlying knowledge base. By using CLUE, we have built a highly object-oriented and declarative user interface. This interface serves as a specification and smooths the transition to other toolkits.

Each presentation is implemented as a CLOS object whose style is described in an associated CLUE contact. The presentation description includes a declarative description of the metamodel relations which are used to establish and link presentation pieces, and the editing and navigation actions (associated either with a presentation piece or the entire presentation). Editing actions result in invocations of evolution transformations which perform the desired changes to the knowledge base.

4.2.1 Implemented presentation styles

As we noted previously, a presentation style captures a particular approach to gathering data, organizing it, and aligning it for display on the screen. For composite styles, no commitments are made about the manner in which inferior items will be displayed or the particular menu options that will be associated with displayed objects.

4.2.1.1 Composite styles

The composite presentation styles which have been implemented for ARIES are as follows:

unbalanced-tree — Tree structures with labeled nodes, e.g., event-taxonomy [Figure 3.1] and decomposition diagrams [Figure 1.3].

flow-diagram — Directed graphs with labeled nodes and edges, e.g., state-transition diagrams [Figure 2.9] and data flow diagrams.

level-list — Presentations of graphical objects in rows—one row for each level of information, e.g., the inheritance structure among folders.

presentation-list — A linear list of objects, e.g., summarizing the contents of a folder - by listing the names and type signatures of all concepts in that folder.

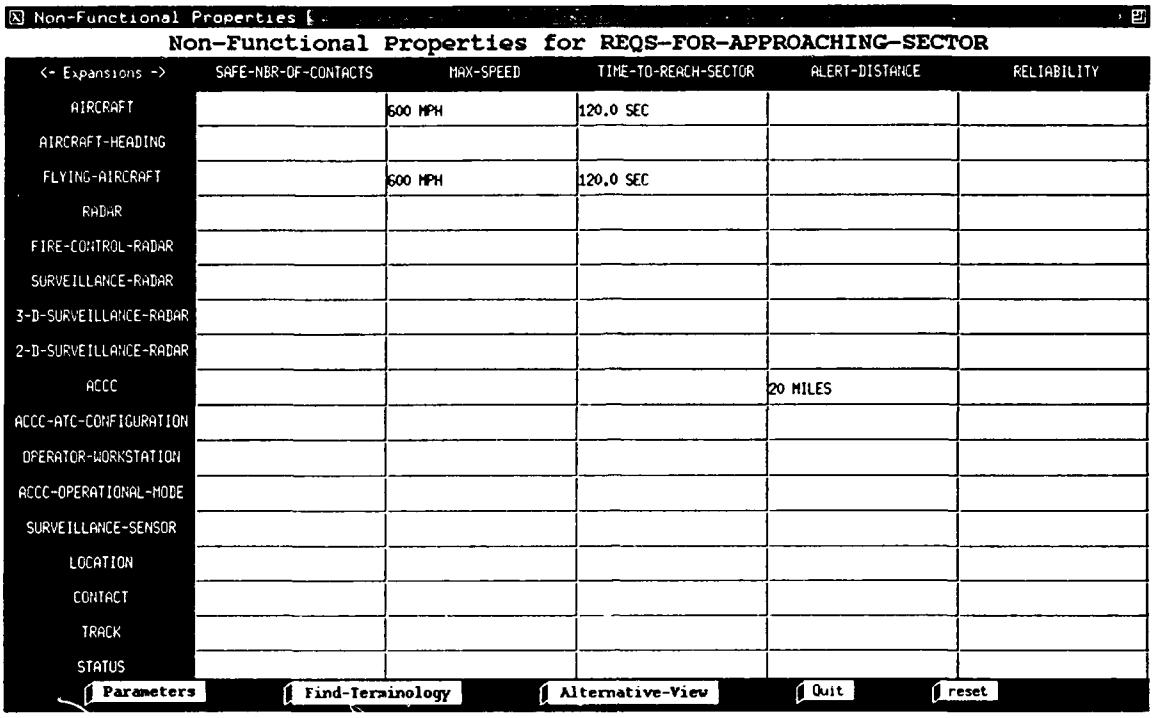

Figure 4.2: Spreadsheet presentation of nonfunctional requirements

matrix — Tabular presentations of data in row and column format, e.g., spreadsheets of nonfunctional requirements [Figure 4.2].

pages — Textual output organized into pages, e.g., the informal (natural language) descriptions associated with a folder; a textual display of the formal Gist specification of the concepts in a folder [Figure 3.2].

4.2.1.2 Individual Components

The portrayal of individual components—boxes, arrows, menu lines—involves icon display and content display. Icons have shape, size, color (or gray scale), and ornamentations. Contents range from very simple name labels to complex textual blocks that are generated through a translation process. Several of the presentations have formal language as their content. *Translation* between the system's internal representation and each of these formal languages is the crucial capability in creating and using such presentations. The textual style (as used in the presentation-list and pages styles) can be in one of the following sub-languages:

English — Concepts of our internal specification language can be described in natural language by our paraphraser tool, as is shown in Figure 2.5. Paraphrasing offers an easy-to-understand portrayal of specification information to those not conversant with our formalisms. This is for output only—the system does *not* do natural language understanding.

Reusable Gist — The behavior-oriented aspects of the specification are represented in Reusable Gist, a version of our in-house specification language Gist that we have used for many years [7]. The system automatically translates its internal representation of specifications into an external Gist form which is much more palatable to human comprehension. This translation goes in both directions—it is also possible to enter or modify requirements and specification information by providing or modifying the external form of Reusable Gist text, which the system parses and translates[2] into its internal representation.

Refine — Our work on software requirements and specifications is part of the larger KBSA project tackling the entire software development process. The formal specifications that emerge from our system will be input to the design activity leading to executable prototypes, and ultimately final implementations. A different group has studied the algorithmic design and coding phases of this development life cycle, and they expect to work with specifications expressed in Refine, a very-high-level language. Thus we have built a translator that outputs specifications in Refine syntax. Of course, there are concepts in our specification language that do not have a direct counterpart in Refine, so it is part of our specification development process to transform specifications using such constructs into specifications that make no such use, prior to production of Refine output.

Loom — we have a translator that, given specification knowledge expressed in Loom (a commonly used knowledge representation language), is able to produce the equivalent in our internal language. Currently this is a only a one-way translation into our representation. We have not constructed the translator to go in the reverse direction, although in principle it should be a straightforward task to do so.

We use Wile's POPART system as the basis for defining translation. POPART, when provided with a BNF-like description of two languages, produces (among other things) a capability for easy definition of translators between those languages (see [79, 78] for details). We use these capabilities extensively.

As a specific illustration of translation, consider again the Reusable Gist expression discussed in Section 3.6:

[2]To define this and other translators, we provide formal definitions of the grammars of our languages to Wile's POPART system, which in return provides us with powerful and convenient language manipulation tools from which we can readily build the translators—see [79] for details of POPART.

```
any aircraft | flight-plan(aircraft,"Los Angeles","New Brunswick")
```

Recall that the internal textual representation of this expression is as follows:

```
(instance-retrieval
    :variable (variable :name aircraft :type entity :determiner any :time present)
    :predicate (query  :concept (reference :name flight-plan :class relation-declaration)
                       :actuals (reference :name aircraft :class instance-declaration)
                                (reference :name "Los Angeles" :class instance-declaration)
                                (reference :name "New Brunswick" :class instance-declaration)
```

Translation of such "instance-retrievals" from the internal form to RG is specified by the following rule:

```
(instance-retrieval :variable !variable#v :predicate !predicate#p)
  ⟹
any !ever-role#v | !predicate#p
```

This translation rule consists of a pattern in the grammar of the internal language, the symbol $\Longrightarrow$, and a pattern in the grammar of the RG language; the ! symbol indicates a pattern variable following (e.g., !variable#v is a pattern variable which can be bound to a parse-tree of the type variable; the suffix, #v, is the means to give this variable the name v). To translate an instance-retrieval, it is matched against the above rule's first pattern; if successful, matching instantiates the pattern variables with the corresponding portions of parse tree in the input expression. The result returned by translation is the second pattern where its instances of pattern variables get their values *by recursively translating* the values bound to the variables of the *same name* in the first pattern. Figure 4.3 illustrates this translation process in operation on our example. The expression to be translated, (instance-retrieval (variable name aircraft ...) (query concept ...)), is matched against the first pattern of the translation rule. This establishes bindings for !variable#v and !predicate#p, namely (variable name aircraft ...) and (query concept ...) respectively. The result of translation is obtained by instantiating the translation rule's second pattern with the variable values computed by recursive translation (i.e., (variable :name aircraft ...) is translated to aircraft, and (query :concept (reference :name flight-plan ...) :actuals ...) to flight-plan(...)). This recursive translation process is a feature of POPART's mechanisms, and saves us the tedium of explicitly coding the recursive translation of substructures in many of the translation rules.

Our complete translator from the internal representation to RG consists of a set of such rules, one for each type of construct in the grammar of the internal representation. We

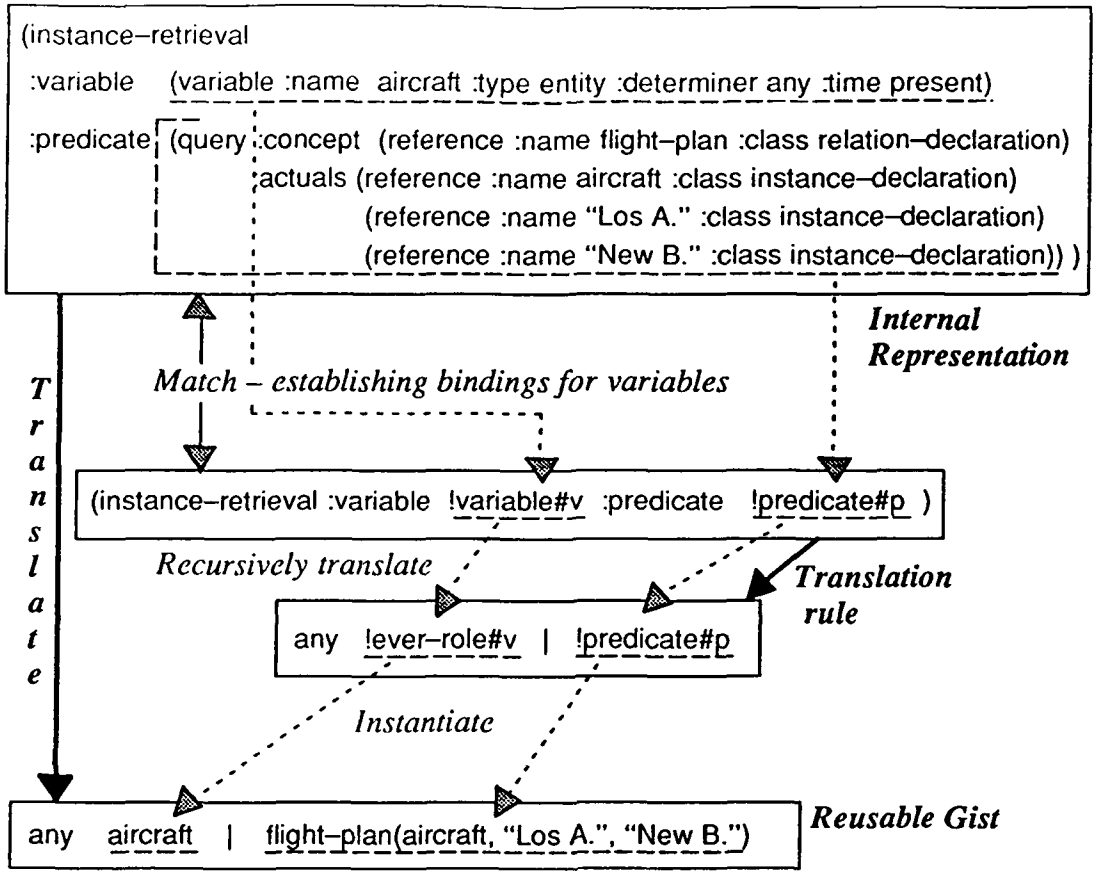

Figure 4.3: Translation from the internal representation to RG

have similar rule sets to do translation in the reverse direction (i.e., from RG to the internal representation), and the other (formal) language-to-language translations of ARIES (namely, from Loom into the internal representation, and from the internal representation into Refine).

In defining the translation rules for producing RG, we invested some extra effort to cause them to use RG's notational abbreviations where possible. By notational abbreviations we mean the syntactic shorthands that permit a more concise expressive form for certain forms of expression, for example, writing addition as in infix expression rather than as a retrieval from a ternary relation ('3 + 4' rather than 'any integer | plus(3,4,integer)'). RG offers a number of such abbreviations, whereas the internal representation eschews these in favor of a more uniform but verbose style. A naive translation of the internal form that did nothing in this direction would, we conjecture, produce unsatisfactory output. We feel that our policy has the advantage of ensuring that RG displayed by the system always makes consistent use of these abbreviations.

We have opted for canonical layouts and translations to provide uniformity and to postpone dealing with some of the engineering issues involved in mixing user-initiated formats with automatically generated formats. If the analyst manually rearranges the layout of a diagram, ARIES maintains data structures which enable it to present the revised layout until modifications to the knowledge base dictate that the presentation be updated. At this point, the user-initiated positions are discarded. A similar capability exists for translations. To preserve the exact form of RG as entered by the analyst would require retention of information that we currently discard (our internal representation does not retain the form of the RG from which it was translated). We have essentially chosen to discard such information; this is obviously a compromise, insofar as we may imagine a more sophisticated system that retained such stylistic choices, but its realization would further complicate the representations[3].

4.2.2 Implemented presentations

Using these presentation styles we have developed 31 presentation types for the ARIES system. A presentation type is defined by a presentation style, a specification of the presentation parts, and a specification of the navigation actions that are appropriate. Figure 4.4 shows the declarative description for one of these presentations. sans serif font shows the actual code. *italic* font shows comments that we have added for this report.

Observe that this declaration provides the information necessary to determine both the information to be extracted from the knowledge base, and how to portray it on the screen.

[3]Furthermore, an *ideal* system might go so far as to recognize the analyst's "style" and be able to present new or changed material in that style!

```
(defpresentation EVENT-TAXONOMY ()
  :name "Event Taxonomy" name of presentation as will appear in menus
  :concept-description "Showing the specializations of an event." menu documentation
  :top-level? T       directs that this appears in top-level menu of presentations
  :grouping 'terminology       subgroup of menu in which this appears
  :method 'projection       currently commentary, i.e., ignored by the system
  :mode 'graphical       currently commentary
  :implementation-model 'unbalanced-tree       style of this presentation
  :do-type 'event-declaration       type of "seed" object being presented
  :po-editors (cons 'up-a-level (top-level-actions))       edit actions available on presentation;
      as well as all the (standard) top level actions, also includes the action to move
      up a level
  :do/po-interface '(,(make-poid
    :po-name :vertex       defines nodes (a.k.a. vertices) of this presentation; more complex
      presentations, e.g., state transition diagrams, also define generation and portrayal of arcs
    :po-type '(shadowed-circle)       choice of how objects (nodes) will be drawn
    :do-type 'event-declaration       type of object being portrayed
    :navigation (standard-tree-navigators)       navigations available on each node
    :do-getter 'generalization       link to follow in knowledge base to extract objects (nodes)
    :direction 'reverse       direction to follow aforementioned link
    :do-namer #'folder-and-relnames)))       function that generates labels for nodes
```

Figure 4.4: Definition of Event Taxonomy presentation

Examples of the former class of information are the declarations that generalization be used, and used in the reverse direction (because we want the *specializations* of the "seed" event), as the relation whose transitive closure is computed to determine the information to be extracted. Examples of the latter are the declarations that select the presentation style (unbalanced-tree), the choice of how to draw individual nodes 9shadowed-circle), and the choice of what navigation menu options are available in response to clicking the middle mouse button on any of the portrayed objects.

The following is a complete list of presentations currently supported in the ARIES system. The presentation types are catalogued as presentations of terminology, folders, behavior, summaries, decomposition, analysis, and formal description. Some of presentations pertain to aspects of ARIES representation and functionality that will be described in later chapters.

4.2.2.1 Presentations of Terminology

Several presentations emphasize the terminology used in specification.

Relation Participation diagrams show all the relations that an object participates in. For example, a diagram might show all the relations which define or restrict radar contacts. This presentation helps analysts determine if all relations on an object type have been defined or if there can be collapsing of relations which represent unnecessary nuances. The style of this presentation is flow-diagram.

Event Participation diagrams show the events that an object participates in. This presentation is intended to be used in much the same way as the relation participation diagram. The style of this presentation is flow-diagram.

CLOS Taxonomy diagrams show a taxonomy of Common Lisp Object System classes. For example, one might be interested in seeing the class inheritance for all implementations, the top class of presentation types. The style of this presentation is unbalanced-tree.

The *Object Type Taxonomy* shows the specializations of an object type. This presentation displays all the type declarations which specialize a given type declaration. It can be particularly helpful to analysts who wish to retrieve the description of an object identifiable only as a specialization of some general concept. In addition, analysts would use this presentation when specifying special terminology. From any of the taxonomy diagrams analysts can access "upper" level terminology. An "up-a-level" button brings up a menu of all generalizations of the displayed concept. Selection of the generalization results in the creation of an object type taxonomy showing specializations from that upper level. The style of this presentation is unbalanced-tree.

Relation Taxonomy presentations show the specializations of a relation. The intended use

of these presentations is similar to the use of object type taxonomies. The style of this presentation is unbalanced-tree.

Event Taxonomy presentations show the specializations of an event. The intended use of these presentations would also be similar to the use of object type taxonomies. The style of this presentation is unbalanced-tree.

Folder Type Hierarchy diagrams showing all the types defined in a folder, and their specialization/generalization hierarchy as lexically defined in the folder. Unlike the object type taxonomy, this presentation does not cross folder boundaries. However, it may present several disconnected trees. These two presentations should be viewed as complementary projections on the underlying collection of type declarations. The style of this presentation is flow-diagram.

4.2.2.2 Folder Presentations

Several presentations display the interaction among folders.

Folders and *Reusable Folders* show the use inheritance structure of all folders or all reusable folders. This presentation can be used to examine the current folder segmentation, the use structure of the current context within the broader picture of all the folders. (Folder structuring mechanisms will be described further in the next chapter.) It might also be used to help find a specific folder to be added to a use list. In all the folder presentations, special icons help the analyst identify reusable folders, the current context, users of the current context, and folders used by the current context. The style of this presentation is level-list. Generally these presentations contain an overwhelming amount of information, however, they are the only presentations which attempt to display (in icon form) the entire knowledge base.

The *Using Folders* presentation shows the transitive use of a specified folder. All folders which trace their heritage to a given folder are displayed. This presentation helps analysts find reusable folders which might contain useful descriptions to be added to a specification. The style of this presentation is coerce-dag->unbalanced-tree.

Folder Inheritance is a more restricted view of the folder world showing only those folders which are used by or use a single specified folder. The style of this presentation is level-list.

Folder Specializations displays specialization hierarchies for folders, as described in Chapter 5.

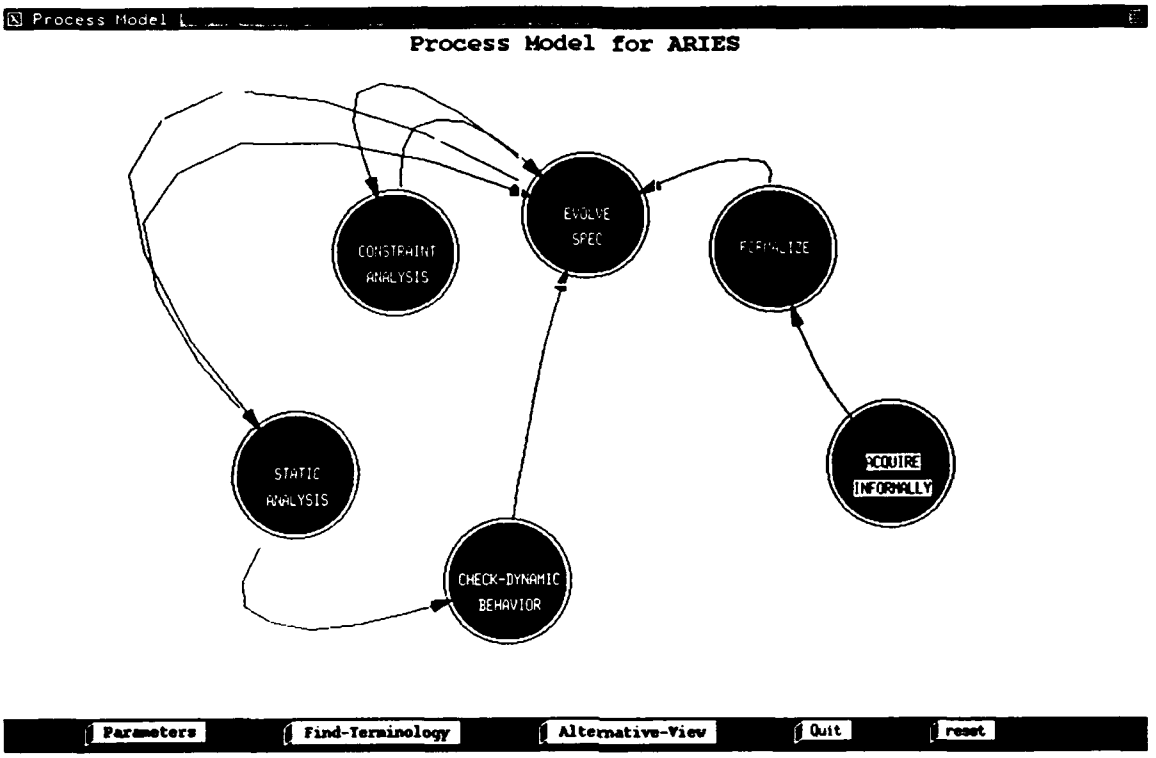

Figure 4.5: ARIES Process Model presentation

4.2.2.3 Presentations of behavior

Some presentations call attention to restrictions on behavior.

The *Aries Process Model* presentation, illustrated in Figure 4.5 is a special presentation of the intended use of ARIES itself. Attached to most process nodes is an activity that ARIES initiates within that process. For example, opening the "check-dynamic-behavior" node brings up the scenario creation environment needed for asking validation questions and running a simulation of a specialized specification (described in Chapter 6. Attached to all process nodes are *examples* which can be followed in instructional mode. In this mode, analysts are guided through typical uses of the ARIES system, as described in Section 4.3.

The *Behavior List* presentation is used to display summary information about the restrictions on behavior--the events and invariants—of a specification. An analyst might use this presentation to get a quick idea about what behavior had been specified.

State Transition Diagram presentations show a collection of states and the transitions (events) which move the system described from one state to another. This presentation helps analysts to specify behavior and ask validation questions. A diagram can be

associated with a particular binary relation that has an enumerable parameter, e.g., a "track-status" relation with a "status" parameter holding all the possible states of an aircraft with respect to a tracking subsystem. Diagrams can also be associated with a folder when the analyst wishes to extract a particular collection of events (displayed as transitions) to achieve validation, review, or acquisition goals. The style of this presentation is flow-diagram.

Information Flow Diagram presentations show the information flow between a collection of events. This presentation can be useful for uncovering missing pieces of a specification (e.g, a fact which is accessed by one event but never asserted by any events). Fact access and modification includes more narrowly defined data flow as a subset. Unfortunately, information flow presentations often display an overwhelming amount of detail about a specification. We designed the consumers, producers, and single event interface presentations to provide more restricted views. Importantly, this presentation points out the need for abstractions which fall between functional/data flow modeling and event/fact flow modeling. The style of this presentation is flow-diagram.

Consumers presentations show all the events which consume information by accessing a declaration. This presentation can be more informative than information flow. It can be used to check on or add to the events which are to consume information. The style of this presentation is flow-diagram.

Producers presentations show all the events which produce information by modifying a declaration. Its intended use is similar to that of the consumers presentation. The style of this presentation is flow-diagram.

Single Event Interface presentations show the information accessed and modified by a single event. They can be viewed as the counterpart to context diagrams used for showing system inputs and outputs. Like consumer and producer presentations, single event interface presentations cut out a more restricted focussed picture of the event world than information flow diagrams. The style of this presentation is flow-diagram.

4.2.2.4 Summary Presentations

Several presentations provide a brief summary of some specification object.

The *Paraphrase* presentation shows a natural language paraphrase of a specification fragment. The paraphrase can help analysts understand the meaning of a specification fragment. In order to validate acquired information, analysts can benefit from different presentations, such as natural language. Figure 2.5 shows the English paraphrase of the specification of the handoff-init event.

The *Documentation String* presentation is a special presentation which is visible whenever the mouse points to a region of the screen which displays a knowledgebase object. It will display textual informal (not Reusable Gist) information associated with the knowledgebase object. This information could be a concept-description, ARIES-documentation, notes-and-warnings, or informal text.

The *Describe Object* presentation displays the ARIES-documentation associated with a particular object.

The *Browser* is a presentation which shows the lexical contents of a folder or specification object. It is an interface to a number of low-level menu-like descriptions of the specification. Analysts may find it useful for quickly navigating around a specification at the level of individual specification fragments.

The *Summary* presentation shows a list of the exported concepts lexically defined in a folder. The style of this presentation is presentation-list.

4.2.2.5 Decomposition

The intent of these presentations is to show decomposition of systems and events.

Event Decomposition diagrams show an internal decomposition of an event into its subevents—the events that are steps of the entire event. This presentation could be used for sketching out the various pieces that make up the body of an event. The style of this presentation is unbalanced-tree.

Functional Decomposition shows the decomposition of a function into its subfunctions. The style of this presentation is unbalanced-tree.

Physical Decomposition shows the decomposition of a system into its physical subsystems. The style of this presentation is unbalanced-tree.

Part Decomposition shows the decomposition of a system into undifferentiated (could be functional or physical) subsystems. The style of this presentation is unbalanced-tree.

4.2.2.6 Analysis Presentations

Some presentations are particularly helpful for analysis of specifications.

Reference Tree presentations show the transitive closure of referenced declarations given an initial declaration. They are useful for declaring the compilation mode for concepts and for identifying places where a scenario can be pruned to obtain a more informative

simulation. The style of this presentation is coerce-dag->unbalanced-tree.

Nonfunctional Properties show the nonfunctional requirements for a collection of system components. Each requirement is shown with an indication (using colors) of its type of support—default, supposition, belief, constant, inherited, or derived through a constraining formula. The style of this presentation is matrix. The rows of the matrix are type declarations or event declarations—components to which one can attach nonfunctional properties. The columns of the matrix are relation declarations which have been marked as nonfunctional-properties associated with a particular folder. For example, processing-time and response-time nonfunctional requirements would be associated with a folder of real time concerns. Accuracy requirements would be associated with a folder concerned with measurement events.

4.2.2.7 Formal Presentations

ARIES provides two presentations of Reusable Gist.

A *Formal Specification* presentation shows the Reusable Gist formal specification of all the components lexically defined in a folder. The style of this presentation is pages.

The *Reusable Gist* presentation shows the Reusable Gist description of a single specification fragment. This description is identical to the description that appears in the formal specification presentation of the fragment's home folder.

4.3 Operational and instructional modes

In the normal operational mode, the ARIES interface permits the analyst to perform any operation or invoke any function at any time. Presentations and their associated functionality provide focus to the analyst's activities, but the analyst is always free to switch to a different presentation, in which different objects are visible and different operations are applicable to them. The underlying reasoning and evolution capabilities are independent of representation, and can be invoked at any time.

This design gives the ARIES user great flexibility, and demonstrates generality of the technologies employed. However, this flexibility is desirable only if the analyst is familiar enough with the system, and with the problem domain, to be able to take advantage of it. Inexperienced users require much more guidance and advice.

ARIES has taken some initial steps toward supporting users with different levels of sophistication. First of all, the Process Model presentation described in Section 4.2.2.3 helps an

analyst keep track of what overall task is currently being performed, and what subsequent tasks are possible. ARIES can provide limited advice regarding actions that one is likely to perform in each state of the process.

The system can be run in a normal operational mode or an instructional mode. In instructional mode, the system helps the user work through examples of ARIES in use. There are currently 13 such examples in the system.

Unlike some computer-based tools with instructional modes what require the user to follow the indicated path through the script, ARIES allows the user to deviate from the example at will. It makes use of a rudimentary plan recognition capability, in which each action performed by the user is matched against a description of the next step in the example. The user is free to perform any action matching the description, affording a degree of flexibility. Furthermore, the user is free to deviate from the script at any time and come back to it at a later point. Depending upon the level of guidance selected, ARIES can simply ignore such detours, or issue warnings when they occur. In any event, a display of the steps in the script is updated as the user progresses, showing the current point in the script, as in Figure 4.6. The display shows a three-level hierarchy of structure in the examples: tasks to be performed (e.g., "Perform a complex evolution), actions to complete as part of the task (e.g., "Find transformation"), and individual items buttons to mouse on or text to type (e.g., "modify-spec"). When the mouse is placed over one of the action descriptions, a detailed description of each input required appears in the documentation at the bottom of the ARIES Manager window. The left column of the display indicates the analyst's progress through the example: "C" indicates that the action has been completed, and an arrow indicates the action that should be completed next.

4.4 Related work

Some CASE tools, such as STATEMATE, support multiple notations. Where ARIES differs from these systems is that in CASE tools the notations are required to convey distinct information, so that edits to one diagram do not result in changes to other diagrams. In ARIES the information conveyed in different presentations may overlap. For example, the Reusable Gist presentation describes many aspects of a system that can also be presented by narrower presentations such as information flow diagrams or state transition diagrams. In this respect ARIES is similar to the the PECAN system, which allows programs to see textual and flow-chart views of programs at the same time [66]. Where ARIES differs is that it allows analysts to edit most of these presentations, and edits to one presentation can result in changes to other presentations. In PECAN only the textual presentation can be edited, and other views are read-only. ARIES makes use of a general mechanism for mapping presentations onto a common underlying representation. This scheme makes

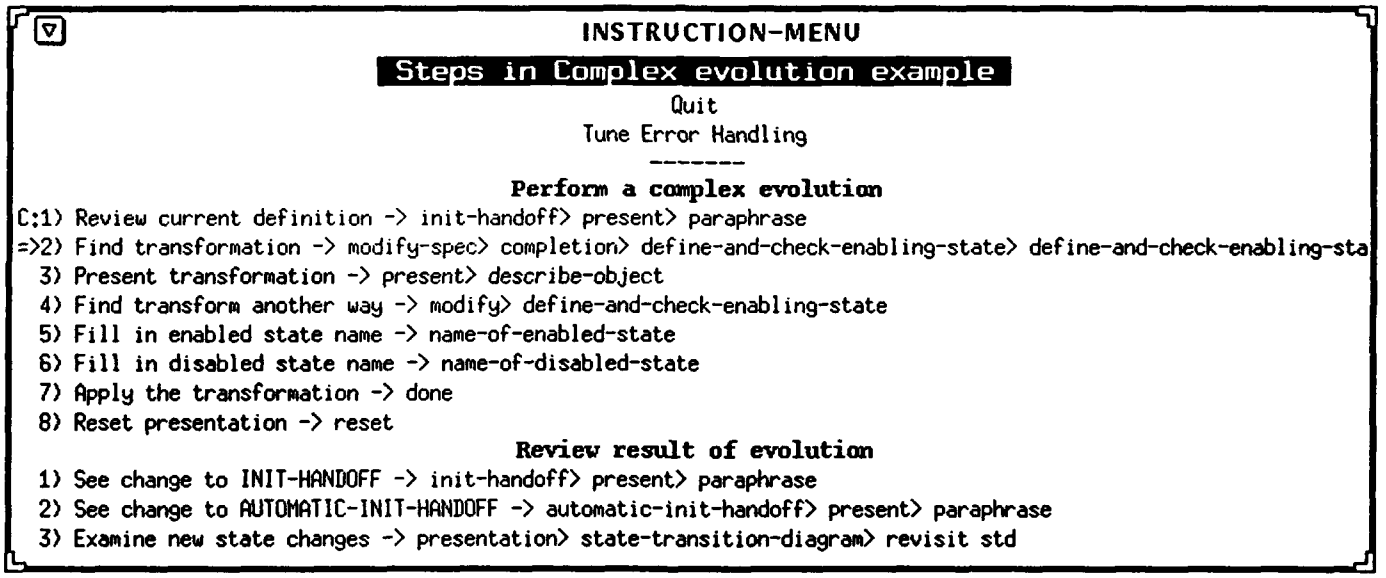

Figure 4.6: An instructional script

it possible to provide multiple editable notations, and map changes onto the underlying representation as well as onto other notations.

The PRISMA project [61] is also a system for assisting in the construction of specifications from requirements. Its main characteristics are:

- Multiple views of the (emerging) specification, where the views that they have explored are data-flow diagrams, entity relationship models, and petri nets.
- Each view is represented in the same underlying semantic-net formalism, yet represents a different aspect of the specification. This representation is suited to graphical presentation and admits to certain consistency and completeness heuristics whose semantics depend on the view being represented (e.g., the lack of an 'input' link in this representation in a data-flow diagram indicates a process lacking inputs; in an entity-relationship diagram it indicates an entity with no attributes; in a petri net diagram it indicates an event with no preconditions (prior events)).
- Heuristics exist to compare the different views of (different aspects of) the same specification, and aid in construction of new views or support checking for partial consistency between views.
- Errors detected by the above heuristics are added to an agenda of tasks requiring resolution, along with advice on how to accomplish that resolution.
- A paraphraser produces natural-language presentations of many of the kinds of information manipulated by the system (e.g., of the requirements information represented in the different views, of the agenda of tasks and advice for performing those tasks, and of the results of the heuristics that detect uses of requirements freedoms).

There is striking similarity between their approach and ours—the use of multiple views and their presentations, and an underlying semantic-net formalism. They have clearly thought about and developed heuristics to operate on or between views, an aspect that we have only recently begun to address. Conversely, we have provided much more support for evolution.

Chapter 5

Cooperative Requirements Analysis and Reuse

There are a number of impediments to smooth cooperation between human and machine in requirements specification work. The system's knowledge base is bound to be incomplete. The analyst's knowledge and understanding is also likely to be incomplete, since otherwise there would be no need to analyze requirements. Thus an assistant may be unable to interpret requirements statements generated by the analyst, and vice versa. Communication between machine and analyst can fail because each is unfamiliar with the names that the other gives to concepts. For example, a common term such as "direction" can mean different things. It may be relative to geographic north, magnetic north, or the current orientation of the object. Without further advice, the machine may misinterpret the analyst's statements, and vice versa. If the assistant proceeds to draw erroneous inferences due to the misinterpretations, the accuracy of the requirements description becomes increasingly suspect.

The problems of misinterpretation and miscommunication also arise when groups of analysts cooperate in analyzing requirements of complex problems. In such situations, different analysts inevitably focus on different parts of the problem. In so doing, they may develop different ways of referring to domain concepts, or make different assumptions about them. They may choose to model the domain in different ways, and make different simplifying assumptions. We contend that these differences in point of view between analysts are a necessary part of the business of requirements analysis. An analysis team can be expected to continually go through a cycle of agreeing on shared results and working independently to refine concepts and explore new territory. Rather than prevent modeling discrepancies between analysts, we wish to develop tools that help to make such discrepancies explicit, and gradually eliminate them during problem analysis.

This chapter describes the knowledge base structuring mechanisms that alleviate communication problems during requirements analysis. By explicitly controlling the degree of sharing between different parts of the knowledge base, we lessen the risk of misinterpretation. Reuse of requirements knowledge is facilitated, without inadvertently introducing knowledge which is in conflict with each analyst's conception of the problem.

5.1 Folders and workspaces

The primary units of knowledge base organization in ARIES are *workspaces* and *folders*. Each analyst interacting with ARIES has one or more private workspaces—collections of system descriptions that are to be interpreted in a common context. Whenever an analyst is working on a problem, it is in the context of a particular workspace. Each workspace consists of a set of folders, each of which contains formal and/or informal definitions of interrelated system terminology or behavior. Analysts can use folders to organize their work in such a way that they share some work and keep some work separate.

The folders can be used to maintain alternative models of concepts, which analysts may choose from when constructing a system description. The move events shown in Figure 3.1 are an example of this. This figure is repeated in Figure 5.1 for ease of reference. These concepts are actually taken from several folders. The names of the folders appear at the bottom of the boxes, and unfortunately are partially obscured in the diagram. Some, such as generic-actions, are not specific to any domain or problem; some, such as land, are specific to aviation. Furthermore, different models are defined within the aviation domain: the maneuver folder models aircraft as moving in continuous trajectories over time, and the basic-descriptive-aircraft-move folder models motion as consisting of straight-line motion from source to destination. Each model is suitable for different purposes. An analyst selects folders by building a new folder that *uses* the folders containing terminology he or she is interested in. Capabilities are provided for locating concepts in related folders, and linking them to the current folder.

The ARIES library of domain and requirements knowledge is also subdivided into folders. The ARIES knowledge base currently contains 122 folders comprising over 1500 concepts. These concepts include precise definitions of concepts, as well as excerpts from published informal documents describing requirements for particular domains, e.g., air traffic control manuals. As an illustration, Figure 5.2 shows the contents of a folder containing some definitions related to aircraft flight.

First note that the folder contains an aggregation of interconnected terminology which analysts access by reference to the folder. Note also that this aircraft folder contains references to concepts (e.g., vehicle, compass-point, symbol) which are not central to aircraft

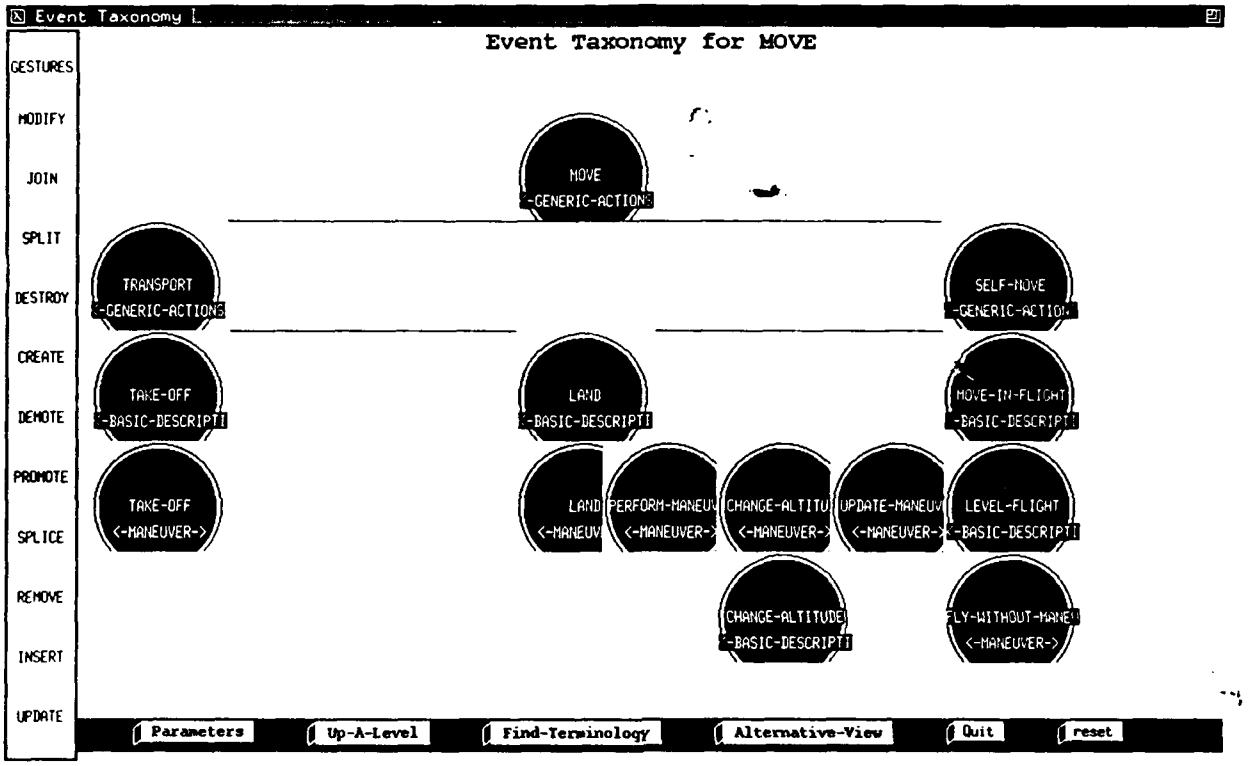

Figure 5.1: Taxonomy of kinds of motion

```
folder AIRCRAFT
exports {
    type aircraft specialization-of vehicle
    type aircraft-heading
       specialization-of compass-point;
    relation in-flight(aircraft);
    defined type flying-aircraft
       specialization-of aircraft
       requires in-flight(flying-aircraft)
    unique relation aircraft-id(aircraft, symbol)
}
uses { VEHICLE;NAVIGATIONAL-DIRECTION;PREDEFINED }
```

Figure 5.2: The aircraft folder

description alone, but are defined in other folders—namely, vehicle, navigational-direction, and predefined. By declaring that these folders are "used" by aircraft, all of the terminology that they contain becomes part of the requirement.

Folders and workspaces have analogues in other approaches to software engineering and knowledge engineering. Workspaces also appear in Terveen's interface to CYC, [76], and serve a similar function, to separate the work of each developer from the work of others. To a first approximation, folders are similar to the package system of Common Lisp [74]: they make it possible to build requirement descriptions out of existing requirement fragments while avoiding name collisions. Like packages, they contain internal and external symbols, and import from other folders. However, packages in Lisp are usually defined for specific systems, and are difficult to combine in a coherent fashion. Folders, on the other hand, are designed to support integration. Another analogue to folders are the specification encapsulation mechanisms of specification languages such as Larch [49], or the "theories" developed by Smith for the KIDS system [70]. As in this other work, we are interested in being able to combine folders in semantically coherent ways, and define mappings between folders. The major differences are that folders can contain both formal and informal information, and are used to organize all knowledge of interest, instead of just abstract data types. Folders in general are thus simply a grouping mechanism, although certain classes of folders have interesting semantic properties. The closest analogue to folders is the notion of "microtheory" in CYC [32]. Both can be used as a general mechanism for organizing knowledge bases, and make it possible to support multiple models of concepts. We have addressed some concerns that have not been addressed in the CYC work, namely how to organize folders to support incremental formalization of knowledge.

Analysts using folders may either perform fine-grained (i.e., individual term) or course-grained (i.e., entire folder) retrieval of reusable requirements. Retrieval of reusable requirements is entirely analyst-directed but is supported in a manner similar to spreading-activation approaches [57, 48, 16]. Analysts can select components which populate any of the numerous taxonomic, entity-relationship, or simple browsing presentations of ARIES. We have emphasized techniques that operate on hierarchies, and techniques that focus search according to the task and problem domain of interest to the analyst. Our approach does not rely on annotating components with keywords (as in Standish's work [72]), or facet values (as in Prieto-Diaz and Freeman's work [64]).

5.2 Folder structuring

Folders are organized in a *use network*. To reuse information from a specified folder, a developer simply asserts that the currently open folder "uses" the specified folder. In a typical requirement development, we would expect to find a large number of intercon-

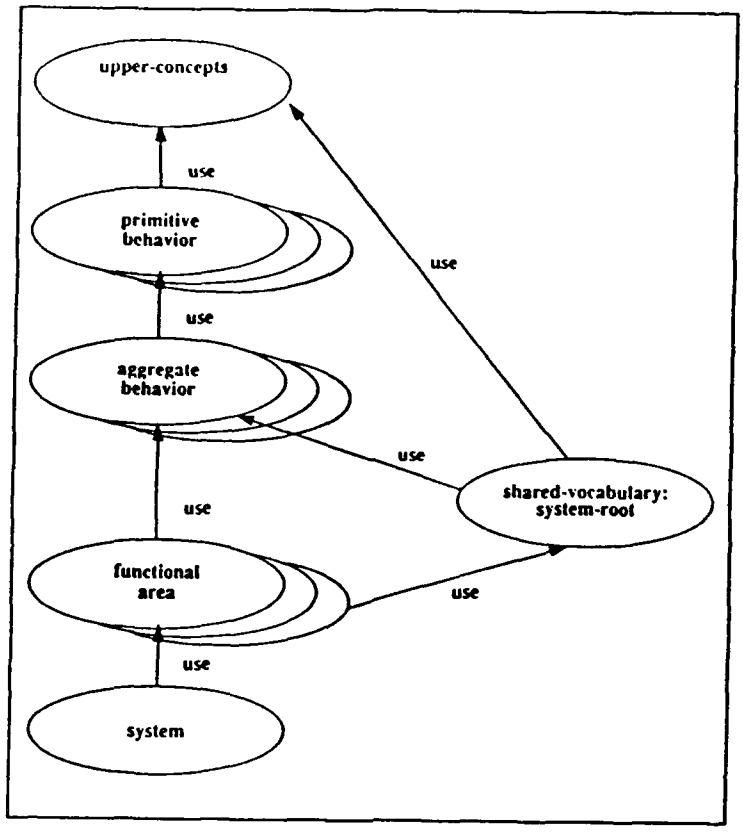

Figure 5.3: Typical relationship among folders

nected folders. Figure 5.3 illustrates the nature of the information typically found in such a network. Systems use the descriptions of functional areas. Functional areas make use of a project-wide shared vocabulary. The project vocabulary uses vocabulary specific to the air traffic control domain, but not specific to any particular project; the domain-specific vocabulary is defined in terms of domain-independent vocabulary that spans across domains. Thus aircraft-tracking uses generic-tracking which requires that locations of objects be known within some tolerance.

A key design issue for the folder framework was whether to make folder use transitive or not. Our approach has been to allow folder consumers the opportunity to individually select folders to use. If an analyst agrees with the model in one folder, it does not follow that he understands the models of all the concepts in the used folders. There are many reasons why this may be so, but the most common is that domain-specific terminology is defined in terms of domain-independent terminology unfamiliar to most analysts. For example, the concept "flight-plan" in air traffic control is defined in terms of the domain-independent concepts "verbal-process" and "artificial-object". "Artificial-object" is part of a generic vocabulary for designed systems, and "verbal-process" is part of the domain-independent ontology for natural language semantics developed by the PENMAN project [8]. Such notions, because they apply across domains, are very abstract and unfamiliar to air traffic control experts. Analysts responsible for codifying domain knowledge will need to understand such domain-independent terminology; individual system analysts will

usually not need to worry about it.

Use structure in itself provides heuristic evidence for folder inclusion: if an analyst mentions a concept that is not present in one of the folders being used, but is present in a folder that one of them use, then it is plausible that that is the concept being referred to. However, it may still be the case that the notion that the analyst has in mind is some variant on what appears in that particular folder. So, it is still useful to keep explicit track of which folders are being used, either explicitly or by inference.

5.3 Reuse techniques

Three further extensions of the folder notion have been developed to support reuse: hierarchies of multiple models, parameterized folders, and the use of higher-order properties. These have been combined in ARIES to provide powerful reuse support.

5.3.1 Representation of multiple models

Analysts may selectively incorporate models of concepts into their requirements, as in the following example. The ARIES knowledge base contains several alternative models for directions: as compass points (e.g., north, south, east, and west), as the number of degrees clockwise from magnetic north, or as multiples of ten degrees from magnetic north (used to mark the direction of runways). Figure 5.4 shows the folders containing these different models.

The most general folder, called direction, includes those properties common to all models of direction. Other folders, named-direction, navigational-direction, and aircraft-direction, define more specific models of direction. These folders are linked in a *folder generalization hierarchy*, shown in Figure 5.5, indicating that the other folders are more specialized models of the same concepts described in the generic folder. An analyst can select from among these models when developing a specification. The selection can also be performed gradually: the analyst may first select the most general, least committed model, and then replace this with a more specific model once the concerns of the specification are better understood.

Our work on supporting multiple models is related to the work on requirements viewpoint resolution [53, 69]. While Liete's emphasis is on the ultimate resolution of discrepancies we have been more focussed on providing the framework for the management and reuse of disparate viewpoints.

folder DIRECTION exports {
    type direction specialization-of ordered-value}
uses {PREDEFINED}
folder NAMED-DIRECTION exports {
    type named-direction specialization-of direction;
    var east:named-direction;
    var north:named-direction;
    var south:named-direction;
    var west:named-direction}
uses {DIRECTION}
folder NAVIGATIONAL-DIRECTION exports {
    type compass-point specialization-of integer specialization-of direction;
    type azimuth specialization-of compass-point;
    invariant compass-range $\forall$ (cp : compass-point) $0 \leq cp \wedge cp \leq 360$ }
uses {DIRECTION;PREDEFINED}
folder AIRCRAFT-DIRECTION exports {
    type runway-orientation specialization-of direction;
    invariant runway-range $\forall$ (ro: runway-orientation) $1 \leq ro$ and $ro \leq 36$}
uses {DIRECTION; PREDEFINED}

Figure 5.4: Folders containing several models for the "direction" concept

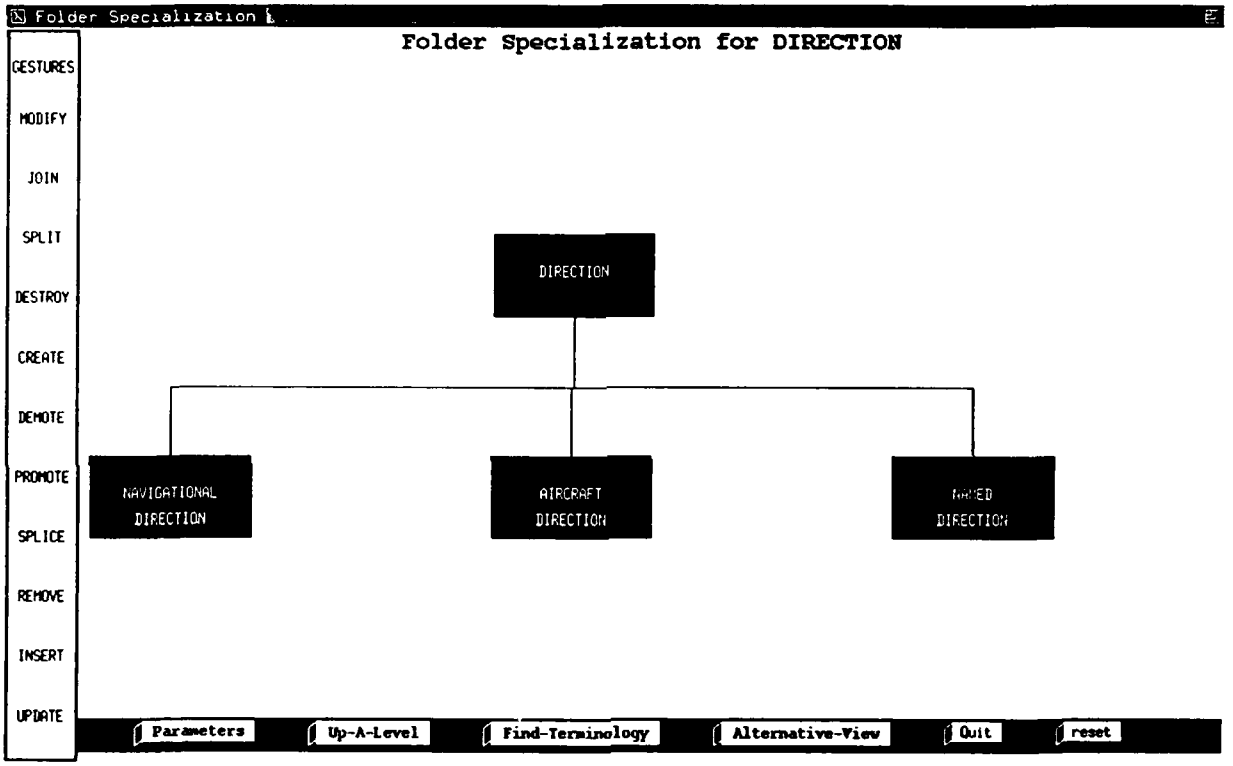

Figure 5.5: Specialization hierarchy of direction folders

5.3.2 Parameterized specifications

Reuse is facilitated through the use of parameterized specifications, i.e., folders that either are parameterized themselves, or contain concept definitions that are parameterized. As explained in Section 3.2.2, parameterized concept definitions contain unbound roles which must be assigned values in order for the concept to be used in a specification. Similarly, folders themselves may have roles, and those roles may be unbound. When an analyst makes use of a parameterized specification, he or she must instantiate a copy of the folder in which the roles are bound. Such parameterized specifications are an important mechanism for representing requirements cliches, and are similar to the approach of the Requirements Apprentice [67].

5.3.3 Reuse through specialization

In ARIES, concepts are organized in specialization hierarchies at the same time that they are organized into folders. The approach to specialization was described earlier in Section 3.2.1. In comparison to many common knowledge representation systems, ARIES makes more extensive use of specialization hierarchies; it supports specialization hierarchies for relations and events as well as types. These extended specialization hierarchies provide more opportunities for knowledge reuse.

5.3.4 Reuse of higher-order properties

Higher-order operators, as described in Section 3.2.3, define properties that hold for classes of concepts. These properties can support reuse: if a concept belongs to a higher-order class, it reuses the attributes that every member of the class exhibits.

An example of a reusable higher-order property is the property of a server satisfying requests in a first-come-first-served fashion. This property is higher order because it is a property that can hold for any process that handles and acts on requests, i.e., it is a property of a particular class of events. First-come-first-served is defined as a temporal relationship between requests and actions, as follows. Let E be the event that is invoked in a first-come-first-served fashion. The generic action of requesting that an action be performed is named **request** in the ARIES knowledge base. Agents will perform **request** actions at various times, requesting that E be performed. In order for a given event E to be first-come-first-served, it must satisfy a temporal constraint between requests and their satisfaction. Namely, if two requests for E are issued, at times t_1 and t_2, and the requested invocations of E occur at times t_3 and t_4, then if $t_1 < t_2$ then $t_3 < t_4$.

5.4 The impact on automated tools

The above techniques all enable analysts to construct new requirements by reusing portions of existing requirements and domain knowledge. Our experience suggests that it is unrealistic to expect all concepts to be used in a requirement to be present in reusable form. Hence, reuse techniques must be complemented with techniques for adapting and modifying existing knowledge. While analysts can informally reuse descriptions (i.e., cutting and pasting as in a text editor), there are many advantages to be gained by using *evolution transformations* to control this process.

There is a strong coupling between parameterized specifications and evolution transformations. Many transformations introduce specification constructs having a stereotypic form. The form of the intermediate object created by such transformations can be stored in a folder and instantiated as needed. We further discuss transformation issues in Chapter 7.

5.5 Examples of Reuse

The following two examples illustrate how ARIES supports the process of reusing requirements. The first example shows how ARIES helps analysts in constructing coherent use lists for folders. The second example shows how ARIES aids the process of integrating folders developed by different analysts.

5.5.1 Adjusting a use list

One of the folders in the Advanced Automation System specification is a folder called automatic-tracking-capability, describing the functional requirements for automatic tracking of aircraft. Figure 5.6 shows the set of folders that this folder uses. The total number of folders used is rather large, fourteen, of which nine are marked as reusable.

Suppose that we try to remove a folder from this list, such as track-data. This can be readily accomplished by performing modifications to the presentation: one clicks the mouse on the icon for automatic-tracking-capability, and a menu of operations of this folder comes up, one of which is REMOVE-FROM-USE-LIST. One then clicks on the folder to remove, and the arrow linking the two folders is deleted. Figure 5.7 shows the result of performing this step.

Now, suppose that we add a new folder to the use list, such as tracker-beliefs, a folder containing generic information deducible from trackers, e.g., the notion that a track signifies a belief that the tracked object is within a given distance from the ostensible location of

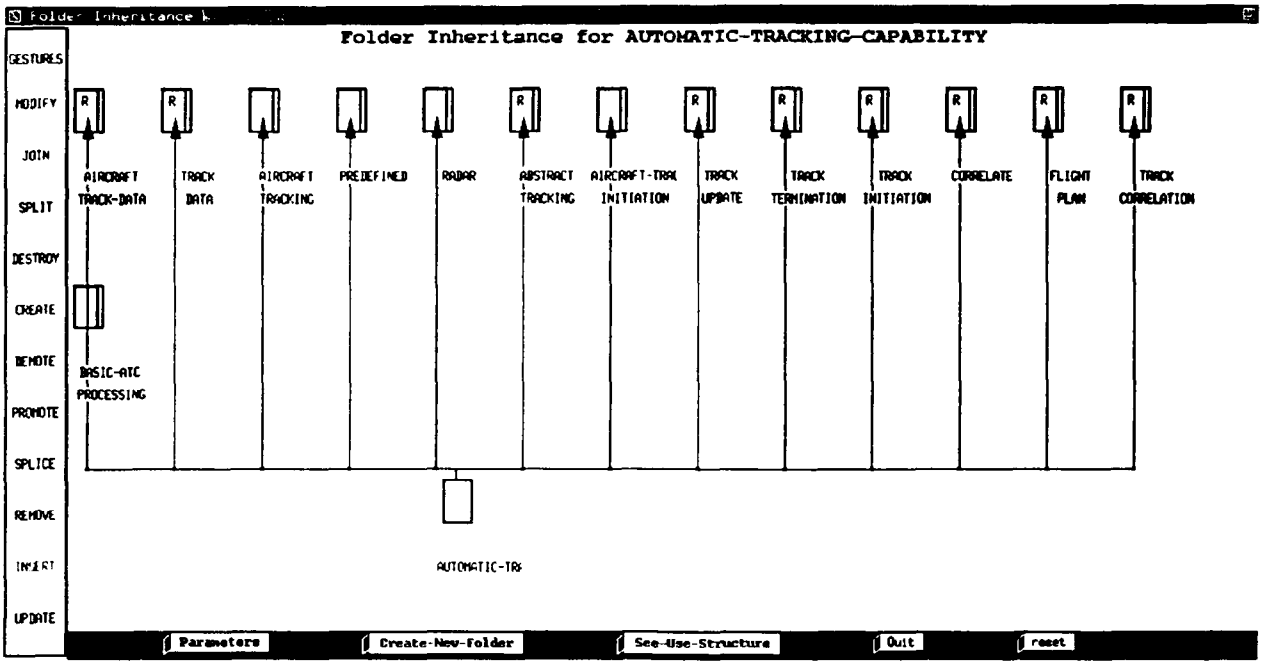

Figure 5.6: Folders used by automatic-tracking-capability

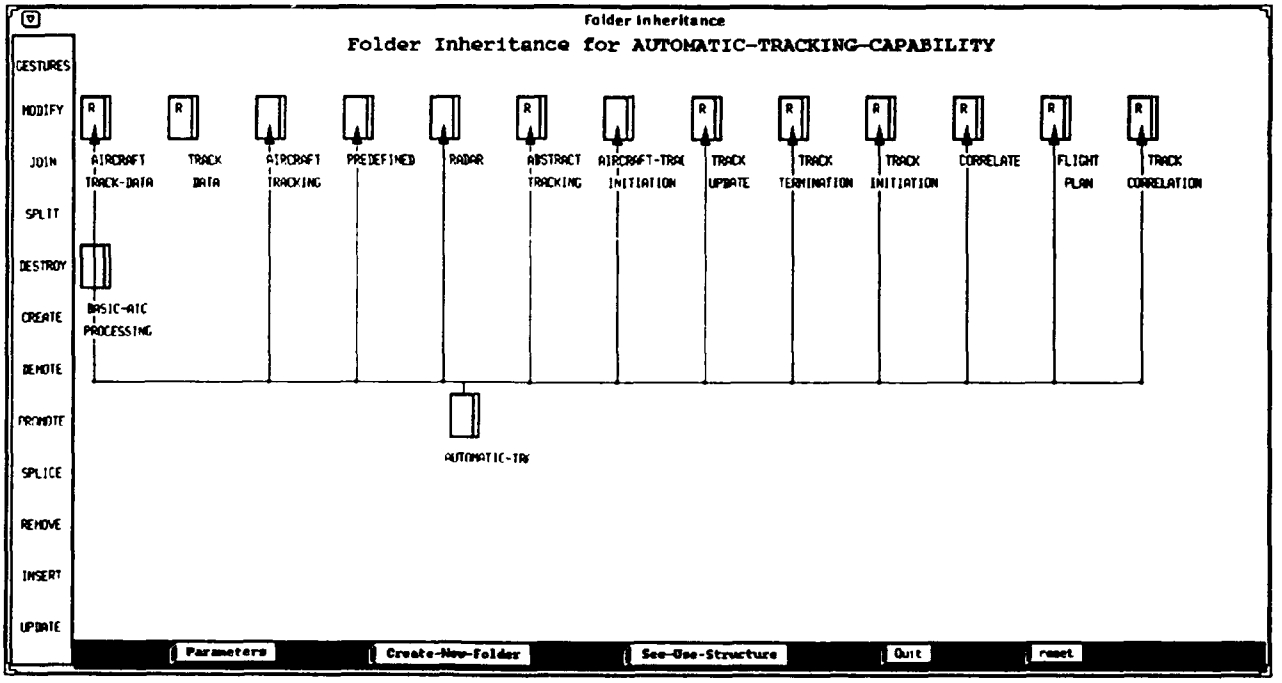

Figure 5.7: Updated use presentation

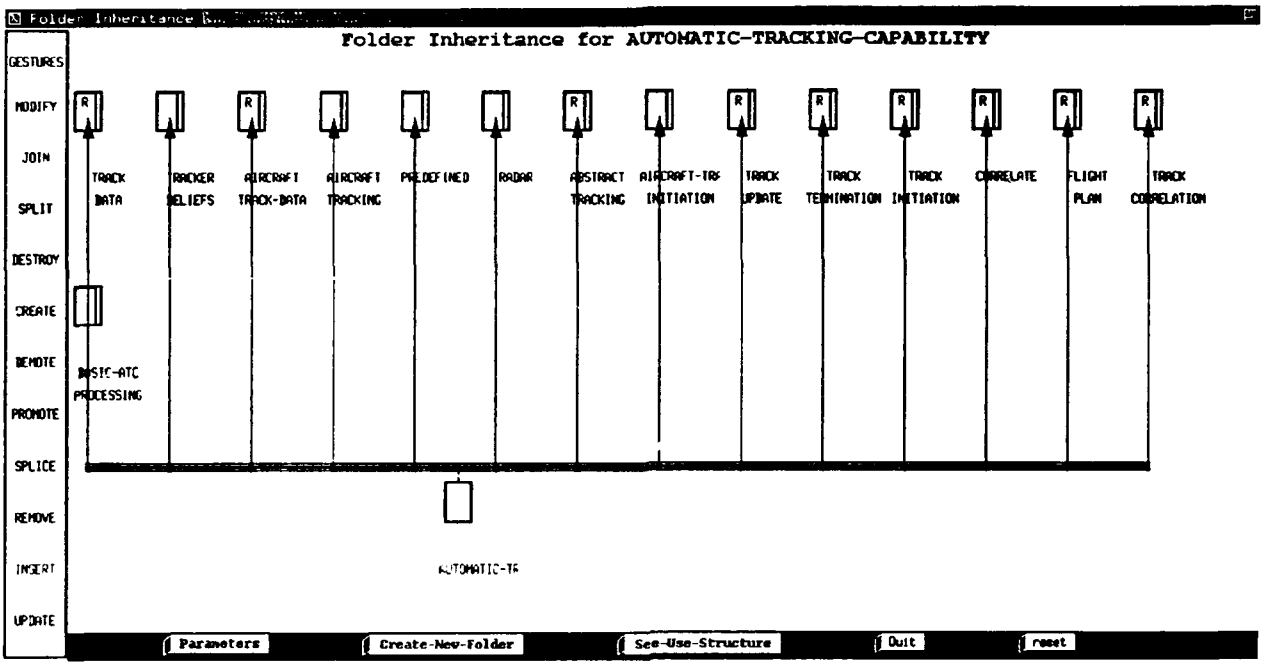

Figure 5.8: Final state of the use list for automatic-tracking-capability

the track, to some degree of confidence. An attempt to add this folder to the use list causes the following automated assistance to be initiated.

- ARIES checks for conflicts between the terminology in tracker-beliefs and the terminology in the folders that automatic-tracking-capability actually uses. In this example, there are no such conflicts.
- ARIES presents a list of folders that the new folder uses, and offers the option of including them at the same time. In this case, it turns out that tracker-beliefs makes use of track-data. We therefore choose to add track-data back into the use list. The final result is shown if Figure 5.8. The changed folders are now marked solid on the diagram, indicating no longer in the use path.

5.5.2 Merging conflicting definitions

As an example of integrating different workspaces, consider again the handoff folder in the Advanced Automation System, discussed in Chapter 2. In order to make sure that the requirements for handoff are consistent, one might want to check to see if there are any conflicting definitions for handoff. One can accomplish this in part by requesting that

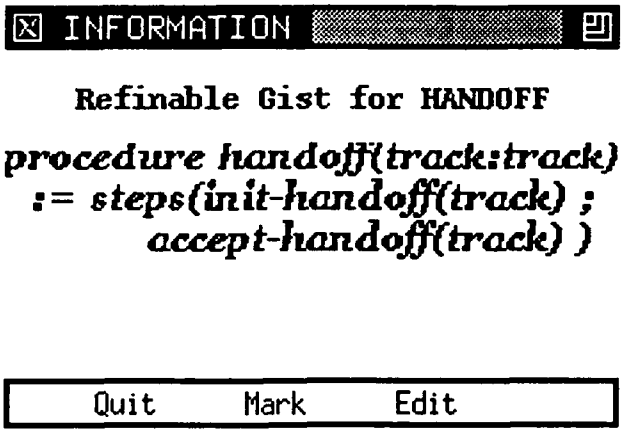

Figure 5.9: Reusable Gist definition of handoff

ARIES search the knowledge base for definitions of events named handoff, and see if there is more than one definition.

It turns out in this example that there two definitions of handoff: one in the handoff folder, and one in the automatic-tracking-capability folder. Evidently the analyst who specified automatic tracking capability was concerned with how handing off of control might interact with tracking.

If we inspect the two definitions, using the Reusable Gist presentation, it appears that they are equivalent. Their definitions both appear as in Figure 5.9. This might suggest that one could simply substitute one definition for the other. However, appearances can be deceiving in cases such as this. Both definitions indicate that handoff consists of two substeps, called init-handoff and accept-handoff. However, we cannot tell by inspection of this presentation whether they are the *same* definitions—there may be duplicate definitions of each of these concepts, just as there were duplicate definitions of handoff. Further analysis is required to determine whether the two definitions can be safely merged.

At this time the ability of ARIES to determine mergeability automatically is rather limited. It can detect attempts to introduce conflicting concepts into a folder, as was illustrated in the preceding example. This enables it to warn the analyst that the two definitions of handoff refer to different definitions of handoff-init. Figures 5.10 and 5.11 show the two definitions of handoff-init, and highlight the main discrepancy between the two definitions: one asserts a handoff-in-progress relation when handoff initiation starts, and the other does not. Presently it is the responsibility of the analyst to recognize such differences, and initiate edits to remove them.

Assuming that the author of the automatic-tracking-capability folder had the same notion

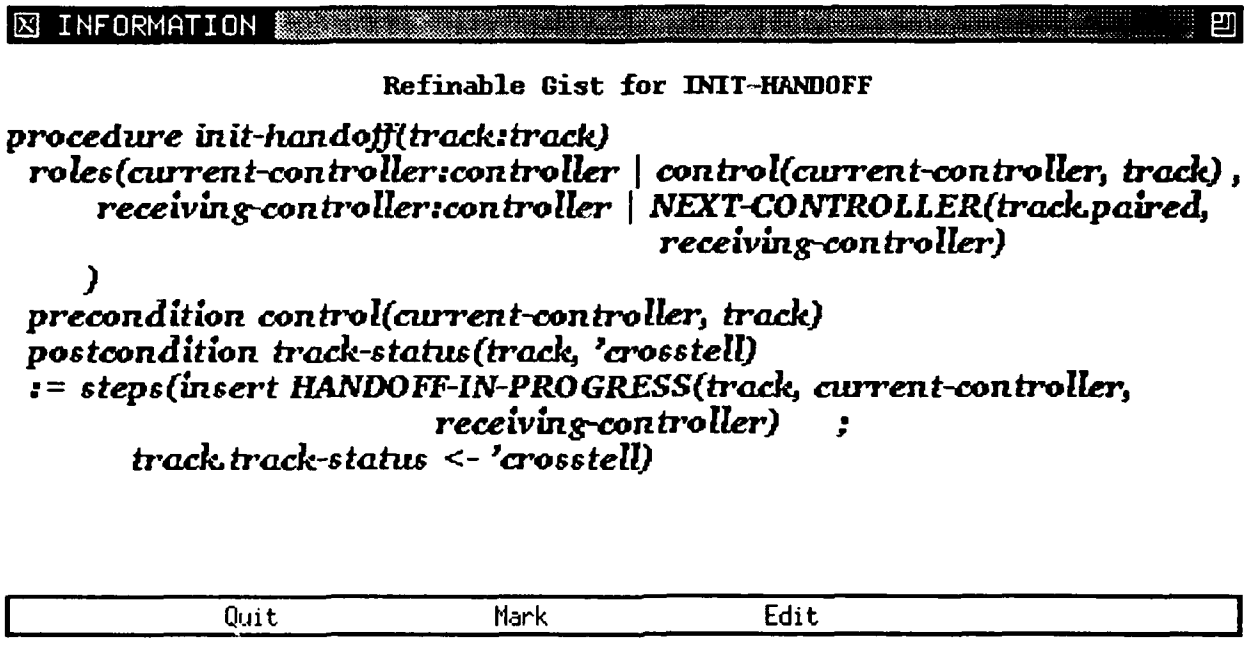

Figure 5.10: Definition of init-handoff in handoff folder

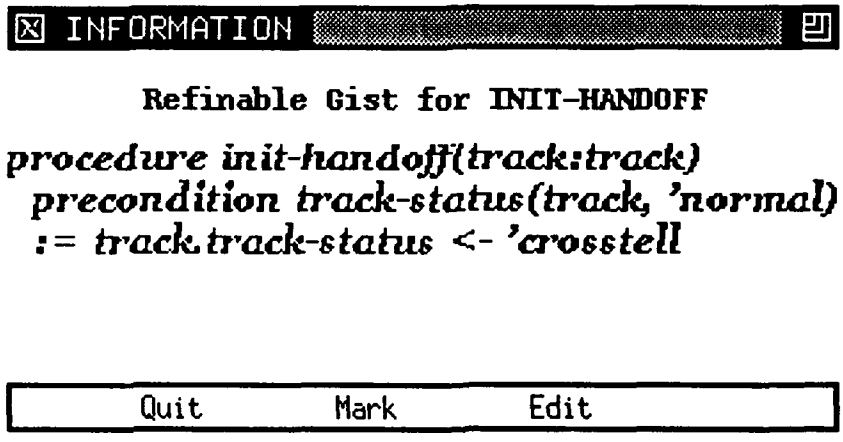

Figure 5.11: Definition of init-handoff in the automatic-tracking-capability folder

of init-handoff in mind as the author of the handoff folder, it should be possible to elaborate the definition in automatic-tracking-capability, and attempt to merge them. ARIES is able to recognize some cases where duplicated definitions really are equivalent, and can be merged, specifically cases of types and relations that have isomorphic definitions, referred to in separate folders or definitions. In such cases the analyst is free to start substituting references to duplicated definitions, and if this is successful, then remove the definition that is no longer needed. In this manner the two definitions of handoff are gradually reconciled and merged.

Chapter 6

Automatic Reasoning for Requirements Engineering

While automatic reasoning in some form goes hand-in-hand with knowledged-based systems, it is instructive to consider separately what automatic reasoning might mean for requirements/specification work and specifically describe what we have implemented in ARIES. Automatic reasoning helps requirements analysts by deriving system/software properties which go beyond original input. By automatically deriving some properties from others, support tools save the analyst from off-line back-of-the-envelope analysis and integrate the results into a unified evolving system description. By recording the reasons for propagated values, a support tool automatically retracts assertions if analysts change their mind about supporting statements.

Automation is crucial because of multiple approaches to system modelling and because of the considerable interdependency that exists among requirements. For example, as was noted in Section 3.5.1.2, there is more than one way of representing states as relations in the ARIES Metamodel. Reasoning capabilities may be required in order to recognize alternative formulations as equivalent. At the domain modeling level, an analyst may specify the sampling rate of a tracker directly or may indirectly constrain the sampling rate by establishing properties of the radar system which supplies the data. System size, power, or accuracy requirements are often in tension with system processing-time requirements. This important nonfunctional requirement area was a focus of the recent Requirements Engineering and Rapid Prototyping Workshop [9]. This workshop pointed to the need for tools which provide for conflict resolution and predictions of the impact of change involving nonfunctional requirements.

In some cases, tool designers can handle the underlying interdependency by requiring that information be acquired and represented in a particular way. For example, as long as states

are created only via ARIES's state transition presentation, we can ensure that states are modeled internally in only one way. However, it is our belief that restrictions such as this are bound to fail. As we argued in Chapter 3, broad expressivity is highly desirable for a requirements acquisition system, and is one of the fundamental design goals of ARIES. Where possible, we have attempted to employ automated reasoning techniques to integrate and reconcile acquired requirements, rather than insist that analysts express requirements in specific ways that ARIES is prepared to accept.

6.1 A framework for reasoning

Our framework for reasoning has been influenced by the nature of human-machine interaction needed for requirements/specification work, and by an analysis of appropriate deduction methods. We have for the most part adopted interactive approaches, where the analyst and the assistant both contribute to the analysis process. The analyst is allowed significant initiative in this process.

When the analyst takes the initiative, and proposes an elaboration or change to a requirements specification, the intelligent assistant should be responsible for monitoring the analyst's change and deriving consequences from the change. The unit of change is a *step* - a quantization of the development process associated with significant analyst-initiated actions to modify the specification. In ARIES, steps are invocations of evolution transformations. Each transformation is typically implemented as a collection of database assertions involving the transformation's input parameters. Thus what appears to the analyst as a single step may in reality consist of a series of steps. An intelligent assistant can provide three levels of support built around the notion of steps: prevention, recording, and consistency maintenance.

- *Prevention*

  Steps are typed and there is a filtering (e.g., not available, not mouse sensitive) of types that are strongly prohibited given the phase of the development, some features of the evolving specification, or the focus of attention (e.g., current folder or presentation). In ARIES, the most important example of this is the filtering of editing gestures. The selection of an editing gesture retrieves a cluster of evolution transformations which will modify the specific objects along the dimensions visible within a specific presentation.

- *Recording*

  Significant (i.e., analyst-initiated) steps are recorded. We anticipate extensive use of step recording for replay wherein the analyst can view the history of these steps, can

retreat to a previous point in the development, and can replay a sequence of steps. Many of these capabilities have already been demonstrated in the earlier Specification Assistant system. In ARIES, data structures are posted when an evolution transformation has been successfully applied.

- *Consistency Maintenance*

  The analyst can anticipate a well-defined protocol for the handling of inconsistency. When a step produces an inconsistent state, an intelligent assistant will notify the analyst of the problem and provide some focussing on ways to resolve the conflict. The underlying implementation approach can be through checking applicability conditions of operators or through forcing a reversible conflict in the specification. In ARIES, some portions of a specification are managed by a constraint propagation system (described below) providing immediate feedback on contradictions. When contradictions occur, ARIES will either automatically resolve the contradiction based on a user-declared level of support for assertions or will immediately notify the analyst of the contradiction and present a list of originating premises which the analyst can select from to resolve the contradiction. As another example, requested modifications may violate applicability conditions of an evolution transformation and ARIES immediately notifies the analyst of the problem. As a third example, concrete simulation may uncover violations of specification invariants. In this case, analysis and feedback to the analyst is delayed until simulation time.

6.1.1 Desirable propagation

Propagation can occur in an aggressive (i.e., resulting in immediate propagation or consistency checking) or lazy (i.e., performed only upon analyst request for information) fashion. For example, automatic classification might typically be performed as an aggressive deduction, while a concrete simulator usually runs in response to a specific analyst request for information. The choice of approach depends on the computational expense to produce information and the relative value of the information to an analyst. The following list indicates some highly desirable forms of propagation associated with requirements/specification support in general.

- Classification—determining the class of an object based on the assertion of several properties that the object participates in. For example, one can automatically classify a tracking procedure as a smoothing procedure (using trajectories to adjust noisy data to conform to known smooth curves) if one knows that the object tracked is a satellite and that reliable tracking is more important than processing time considerations.

- Customization—applying a general principle to a specific situation. (e.g., $\forall(x)P(x)$ —> $P(x_0)$ when only one x exists.) For example, a loop specification that applies a particular procedure to each entry in an enumerated list can be simplified when the list contains only one element.

- Defeasible Reasoning—supplying default values that can be assumed to hold in the absence of evidence to the contrary. Specification of user interfaces can take advantage of default values for positioning and extent of windows. Nonfunctional properties can be assumed to have a stringent value unless otherwise specified.

- Concrete Simulations of System Behavior. For example, a simulation might show the behavior of traffic lights when individual vehicles are located in traffic lanes and oriented to move toward a traffic intersection.

- Logical Entailment—deduction of logical consequences of assertions. For example, if a relation R is known to be transitive, $R(a, b)$ and $R(b, c)$ justify the conclusion that $R(a, c)$.

- Symbolic Evaluation. Symbolic evaluation is a form of testing. Rather than executing a program with respect to data values, symbolic data is used. As a program is executed, symbolic expressions are built based on the statements executed. At the end of an execution, the output value from a symbolically evaluated program is some symbolic expression over the input symbols.

- Coercion—automatically supplying specification fragments that fill in the gaps between other fragments. For example, reusable components may contain parameters which can be filled in with parameters from other components discovered through an examination of an approximation space. As illustration, the aircraft location associated with a tracking specification can be supplied with slant range, azimuth, and altitude measurements generated by a radar component.

- Explanation—tracing facts back to original assertions can help an analyst to understand why specific requirements have been established. In conventional developments, requirements traceability matrices provide a text-oriented form of syntactic explanation (e.g., a paragraph number from a document, an allocation to a subsystem, the name of the specification author). With deduction we aim at semantically based explanations using underlying formalisms and paraphrasing capabilities to provide more informed descriptions of requirements and requirement interrelationships.

- Retraction—removing all facts which are no longer supported. For example, a processing time requirement may restrict a systems ability to meet an accuracy requirement. If the processing time requirement no longer holds, the restriction on accuracy should also be removed.

6.1.2 Tractable computation

It is important to control the application of such reasoning techniques to gain as much power as possible while avoiding intractable computation. The ARIES representation is expressive enough that deriving useful abstractions of specifications can be computationally expensive or even intractable. A case in point is the specialization hierarchy of events. Events are linked to other events by the generalization relation, just as types and relations are. Once such relations have been computed, presentations and transformations can take advantage of them. But our definition of event specialization requires that the preconditions and postconditions of the generalized event must be implied by the definition of the specialized event. Since preconditions and postconditions can be arbitrary logical predicates, perhaps containing temporal or higher-order operators, proving that such a specialization relation exists can be arbitrarily difficult. To avoid intractable reasoning, we have focused on more specialized techniques that can be opportunistically employed to answer questions at critical times in a specification development. ARIES processing elements need only operate on a narrow subset of the knowledge base. The fact that one relation may be defined in terms of others is not apparent to the tools. Thus each tool can view the knowledge base as if it had narrow expressivity.

6.2 Approaches to reasoning in ARIES

Reasoning capabilities in ARIES fall into two categories. The first category consists of reasoning that is performed automatically in reaction to changes initiated by the analyst, or in response to queries posed of the knowledge base by tools. The abstract relations in the ARIES Metamodel, which are computed automatically as described in Section 3.5.1, fall into this first category. The second category consists of tools which analysts may invoke directly in order to draw conclusions about the specification. This category includes a simulation tool. A compiler generates executable code from ARIES specifications, and this code is then run in the simulator. The simulator notifies the analyst whenever a requirement is violated during the simulation run.

While it is possible to uniformly handle all invariants by testing them in simulations, there are advantages to having the ability to reason about invariants as static properties of systems, as requirements are being acquired. In order to use a simulator to validate requirements, it is necessary to set up a dynamic environment to perform the test. Other general purpose validation techniques, such as theorem proving, can require substantial processing time and can be difficult to use. Low-cost analysis techniques that can be employed automatically by the assistant can reduce the amount of effort analysts have to spend with general-purpose tools.

Static analysis techniques rely on special-purpose algorithms to perform specific kinds of analysis tasks. They can be thought of as checking constraints, for a special class of constraints. This point is developed in some detail in [77].[1] Static analysis techniques are employed in ARIES both in reactive constraint processing tools and in analysis tools invoked directly by analysts.

6.3 Automatic constraint analysis

6.3.1 Constraint propagation

By modeling redundancy within a structural framework for specifications, we exploit constraint propagation in a manner similar to its use in circuit design [71]. In the ARIES prototype, this constraint capability has been primarily used for enforcing nonfunctional requirements and for managing complex mathematical or domain-dependent engineering interrelationships.

The approach we have taken is similar to other constraint propagation work. As with other approaches, each constraint restricts the values for a collection of variables. The restrictions are often bi-directional (e.g., in the equation $a + b = c$, any two assignments restrict the third). The constraints must be between different variables, e.g., constraints such as $a = a$ are not permitted.

Our approach differs from constraint satisfaction methods, since we are not typically tasking ARIES with finding a complete solution. In the requirements domain, underconstrained situations are the norm. Analysts "fill in" enough information to properly restrict subsequent implementations; completeness is difficult to achieve and often unnecessary. Our concern is centered more on detecting those conflicts which arise when analysts inadvertently specify unachievable systems.

6.3.1.1 An example

An air traffic control example may be helpful to clarify the issues involved. To ensure that air safety continues during handoff, air traffic control specifiers may require that aircraft are tracked for a minimum time or minimum distance before entering an air sector. They may also require that a minimum number of radar contacts are made on entering aircraft. These

[1] This paper describes how a constraint system might be used to characterize and enforce new classes of constraints. The following are several classes: extended type checking, dimensional analysis, reasoning about units, and estimating error bounds.

requirements are related to other factors which may be beyond the control of the specifiers such as maximum speed of aircraft and scan period of radar installations. Typically, these relationships are expressible as mathematical or engineering approximation formulas. One such formula was shown in Figure 3.2. The following is another example of a similar nature:

```
invariant inv2 Forall (ACCC:ACCC, ac:aircraft)
    time-to-reach-sector(ac) * max-speed(ac) =
        alert-distance(accc)
```

At any time in the specification process, analysts need to know what parameters might impinge on a given requirement. For example, the analyst might wish to know something about the relationship between aircraft speed and the rest of a specification. Also, analysts need to know about conflicts that exist between competing requirements. For example, if the aircraft speed, the radar scan-period, the safe number of contacts and the time to reach a sector are all set independently, the specification will be overconstrained (the time and distance values follow from the other parameters).

6.3.1.2 ARIES constraint mechanisms

ARIES implements a constraint system modelled on Steele's Constraint Language [73]. This system performs local propagation of values and various forms of consistency checking. Propagation occurs bi-directionally through propagation rules connected to nodes in constraint networks. An underlying truth maintenance system is responsible for contradiction detection, retraction, and tracing facts back to originating assertions.

The significance of the ARIES mechanisms is that they blend constraint-based reasoning with other powerful reasoning capabilities to form a substantial basis for requirements/specification support. We have used a hybrid implementation approach similar to the Socle-based [38] approach of KBRA. In Socle, variables are slots of frames and constraint processing complements frame-based inheritance and procedural attachment. In ARIES, variables are relations on type-declarations and constraint processing complements AP5 consistency and automation rules.

Associated with a relation (e.g., scan-period) and a type declaration (e.g., radar) there exists a *justification* structure. AP5 adders and deleters are used to push information across the AP5/Constraint boundary to the justification. The elements of the justification structures are described fully in [73]. Briefly, a *cell* is created for a variable (in ARIES, a relation and type declaration pair). This cell is a member of a *propagation node*, a collection of cells and a repository. Other cells in the node are *pins* of the constraints which restrict

the variable. For example, the scan-period x radar cell will be in a node with a pin of a multiplier constraint (inv1) and an equate constraint (inv3). The repository contains information shared by all the cells including the supplier cell—the cell which carried the value to the node, the rule which fired to carry the value, the support level (default, supposition, belief, or constant), and the value itself. If the cell is directly associated with a variable, it contains back references to that variable. In all, a node contains enough information to locally propagate or retract the results of change. When a node gets a new value, all the pins in the node are examined and associated rules are fired.

The four levels of support—default, supposition, belief, and constant—are set by the user to model the volatility of an asserted value. ARIES uses this level of support to either automatically retract culprits when conflicts occur or to at least notify the user of the most likely culprits for retraction. Only weakest level premises are reported to the user (i.e., intermediate values in complex formulas and values with higher support levels are pruned). In the case of conflicts, defaults automatically bow out. Somewhat tentative values such as results of measurements which may need to be checked again or guesses can be asserted with supposition support. A strongly believed value can be asserted with belief support. In addition, belief support can be used to layer the ordering in which users must deal with conflicts. An assumption about the appropriateness of making a computation can be achieved by setting an activator/inhibitor pin (available on many constraint types) to activate with belief status. The inhibiting propagation option is then hidden until the user gains confidence in tentative results and the conflict exists among competing beliefs. Finally, analysts can enter never changing value (e.g., π) as constants.

6.3.1.3 Helping the analyst manage nonfunctional requirements

The ARIES user interface to the constraint subsystem is through the Nonfunctional Properties presentation. Some requirements are naturally acquired via such speadsheets. Nonfunctional requirements are often of this character. Figure 4.2 shows a spreadsheet of nonfunctional requirements related to performance of radars monitoring the approach to an airspace sector.

The "fall through the cracks" problem is addressed by a relation which keeps track of the specific properties which are to appear in the spread-sheet for a specific system context (i.e., a folder). For example, reliability, scan-period, alert-distance, time-to-reach-sector, max-speed, safe-nbr-of-contacts, sampling-interval, sensor-error, max-acceleration, and positional-error appear in the atc-tracking-reqs spread-sheet presentation. By inspecting this presentation, the analyst can see if all concerns have be dealt with. More directed agenda-like queries such as "Tell me all of the atc-tracking properties that I care about, but have not yet defined" can be easily formulated, but at present are not part of the user interface.

ARIES provides for some unit analysis capabilities. Nonfunctional properties have unit-types. For example, the max-acceleration relation is an *accel-unit* type. This implies that allowable values include only values whose units are of the form distance-unit divided by time-unit squared (e.g., ft/sec^2). These unit types are used to check data entry and also to perform dimensional analysis and unit conversions. Qualitative unit types are also provided. These types have "acceptable values" which are selectable from menus when the analyst attempts to set a value.

6.3.2 Incremental static analysis

The other two major types of dynamic reasoning in ARIES are derivation of abstract AP5 relations between specification components, and incremental derivation of static analysis information. Derivation of abstract relations has already been described in Chapter 3. Incremental static analysis will be described in more detail here.

The ARIES incremental static analysis package is derived from the static analyzer developed for the Specification Assistant [47]. Like the earlier system, it incrementally maintains an index between each definition in the specification and the references to each specification. In the process it builds and maintains a symbol table with entries for each definition. This information is employed by further static analysis tools which may be invoked by the analyst as needed.

The ARIES incremental analyzer improves on the earlier system in several respects. The earlier incremental analyzer did relatively little analysis in an incremental fashion. Rather, it would record when static analysis information became out of date, due to modifications to the specification. If a tool such as a transformation needed the static analysis information, and the current information was out of date, it would have to be recomputed. This resulted in a noticeable delay in processing.

The new incremental analyzer actually maintains the static analysis information incremenally. It is triggered by AP5 rules in response to any change to the AP5 relations in the ARIES knowledge base. Furthermore, it maintains more useful information incrementally. For example, it associates with each folder a list of the references in the folder that do not have a corresponding definition. This list is updated as definitions or references are added, modified, or deleted. This improved analysis is achieved without incurring a performance penalty.

The reason why the static analyzer is able to do a better job is that it operates on the ARIES internal representation, while the Specification Assistant's analyzer operated on Gist parse trees. Because Gist syntax and semantics are quite complex, analysis was time consuming, and incremental analysis was difficult. The ARIES representation is substantially simpler,

with fewer constructs and a more straightforward semantic interpretation. Incremental processing is therefore much easier to achieve.

6.4 Static analysis tools

Various analyst-invocable tools have been provided which aid in the analysis process. Some of these are specialized static analysis tools, making use of information gathered and maintained by the incremental static analysis process. The other major analysis tool is the ARIES Simulation Component, which is described in the next section.

6.4.1 Static analysis tools

Several static analysis facilities are provided for analysts in the form of functions that may be invoked on folders or folder components.

The most commonly used static analysis facility is the type checker. It reports errors such as unresolved references, duplicate definitions, type mismatches in expressions, and the like. These sorts of analyses have been well documented elsewhere [1]. The main difference between the ARIES type checker and other similar systems is its reliance on the information gathered by the incremental static analyzer. The incremental analyzer has already identified undefined references before the type checker is invoked; the type checker reports these errors, and then proceeds with further analysis. Subsequent analysis relies heavily upon the symbol table that has already been constructed.

Other kinds of specialized analysis is provided as part of the type checking package. Executability analysis checks for the constructs that cannot be executed by the simulator, either high-level specification constructs or incomplete definitions. Mergeability analysis compares definitions that an analyst wishes to merge into a single definition, and identifies incompatibilities that could prevent such a merger. A range of specialized analyses can be potentially provided in this fashion.

6.5 ARIES Simulation Component

Simulation in ARIES is performed by the ARIES Simulation Component (ASC). ASC is used by analysts to reason about system behavior. In particular, the goal of ASC is both to uncover undesirable behaviors and to ensure the presence of desirable behaviors in the

current specification. The result of this analysis is either to raise the analyst's confidence in the correctness of the specification or to provide motivation for modifying the specification.

A variety of specification approaches have relied on direct execution or simulation to reveal system behaviors. Guindon in [33] well documents her observations of system designers at work and the benefits they gain from performing mental simulations of domain scenarios during the development process (i.e., during requirements acquisition, specification development, and design). Below is an adapted list of the benefits noted by Guindon:

- Analysts gain a better understanding of stated requirements.
- Analysts opportunistically uncover superfluous and/or missing requirements.
- Analysts can evaluate the current specification and design against stated requirements and domain scenarios.

Though the designers Guindon studied achieved these benefits during mental simulation, Guindon advocates them as justification for automated support of simulation in general. It is toward this end that the executable specification community has been working for some years. Systems within this community include: Software Refinery, PAISLey, Statemate, and OBJ. In general, each of these systems allows the analyst to execute a specification in order to reveal the specification's behavior. The problem is that these systems are limited in their ability to execute specifications which are inconsistent, incomplete, and/or ambiguous. When an analysts using an executable specification approach simulates his/her specification it will break at the first problem encountered. This is not necessarily bad since it does act as a pointer to a problem in the specification, but it may not be the most desirable result since the analyst may be focusing his/her activities on another part of the specification. The analyst may not wish to deal with this first problem now, but is forced to in order to achieve executability. This is in contrast to Guindon's designers which because they are performing mental simulations are able to finesse these problems, focusing on the behaviors of specific parts of the specification which are ready for simulation (i.e., believed to be consistent, complete, and unambiguous) while ignoring or approximating the rest of the specification.

Currently an analyst using an executable specification systems handles these problems by using the same techniques which mental simulation is able to take advantage of, in particular: focusing, approximating, and ignoring. They do this informally by building a simplified specification which focuses on specific parts of a more complex specification while approximating other parts of the specification. Though analysts have had some success with this approach, it has been done in an ad hoc fashion without explicit identification of the analyst's focus and without justification of why some concepts are approximated while others are ignored and others are left unchanged. As a result, one can not formally

characterize the relationship between the original specification and the resulting simplified specification. This in turn must cause one to question the validity of a simplified specification's execution as an accurate predictor of behaviors in the original specification.

ASC differs from the systems above by formalizing the above notions, i.e., making its focus explicit and justifying the approximations it introduces. As a result, ASC is able to formally characterize the relationship between the original specification and the resulting simplified specification and in turn the analyst is able to use the simplified specification's execution as an accurate predictor of behaviors in the original specification. This last characteristic allows the analyst to infer the results of reasoning performed on the simplified specification about the original specification.

An understanding of how ASC accomplishes the above stated goal begins with the identification of domain scenarios. Guindon in [33] describes how designers used scenarios as test cases during mental simulation. In this context, scenarios where used to determine whether a particular projected chain of events occurred or not. In ASC this is known as a **validation question**. In mental simulation, the designer uses his/her intuition about the specification and the scenario to determine what concepts in the specification were relevant for mental simulation. Depending on the relevance, the analyst decides either to include, approximate, or ignore a concept and its associated behaviors in the mental simulation. ASC accomplishes this same task by expressing validation questions formally as a modified Data Path-expressions [39] and by performing influence analysis on the specification and validation question to determine the relevance of individual concepts. ASC then uses the results of this analysis to recommend to the analyst approximations which can be introduced to create a **simplified specification**. The introduction of approximations is a reformulation process. It continues until the simplified specification corresponds to the informal focus referred to during mental simulation. At this point the simplified specification is simulated to answer the validation question. More precisely answer the question, "Is the data path-expression realizable in the simplified specification?" This answer is then inferred about the original specification. The end result is that the discovered answer either gives the analyst greater confidence in the correctness of the specification or motivates the analyst to modify the specification.

6.5.1 Validation questions

One of the implications of Guindon's study [33] was the importance of scenarios as a means for expressing problem domain knowledge. This idea is further supported by Fickas and Nagarajan in [26] where they describe how analysts use hypothetical examples to explain concepts, argue for/against the inclusion of concepts, and further refine concepts in the current specification.

An interesting feature about these scenarios is their inherit completeness. When a domain expert describes a scenario to an analyst he/she is relating a complete snapshot of the behaviors the domain expert is concerned about. Via scenarios, a domain expert can describe specific situations (whether normal case or exceptional case) at any level of detail and in terms of the domain observable behaviors familiar to the domain expert.

Similar to how scenarios where used as test cases during mental simulation, ASC uses scenarios as redundant descriptions of system behavior. The goal of simulation is to confirm that the scenario is really redundant and that the behavior described by the scenario is implied by the specification. It is in this light that Data Path-Expressions seem an appropriate notations for formalizing scenarios.

6.5.1.1 Data Path-Expressions as Scenarios

Data path-expressions (DPEs) are an extension by Hseush and Kaiser [39] of generalized-path expressions [31] which in turn trace their lineage to path-expressions [13]. In general, DPEs are a means or characterizing execution paths through the behavior space of a program via event patterns. DPEs have been used by data-oriented debuggers to monitor the execution of a program and detect when the current execution flow is or is not the same as the path described by DPE pattern. Such a pattern describes a partial ordering of events and is not intended to describe a total program trace (i.e., arbitrary events may be interspersed in a execution path which eventually satisfies some specific DPE). DPEs, as described here, have two type of operands and a powerful set of operators for specifying the concurrent characteristics among events. The following is an abbreviated description of DPE syntax and semantics.

The syntax of DPEs corresponds to regular expressions. The operands are either control events or data events.

- A control event corresponds to the invocation of the named procedure or demon or any specialization thereof. For example, *Move* is a control event which is satisfied whenever any of the following occur: 1) the procedure *move* is invoked, 2) the procedure *self-move*, a specialization of *move*, is invoked, or 3) the trigger for the demon *continuous-move*, a specialization of *move*, is satisfied (and thus in a sense, *continuous-move* is invoked).
- A data event corresponds to a database state becoming true and is determined at the end of each atomic update. This state can be any predicate and is enclosed in square brackets. For example, *[signal-color(tl-1, 'red)]*.

The operators in DPEs are:

- exclusive or (+) – exactly one associated event occurs
- partial concurrency (&) – two associated events are partially concurrent
- sequence (;) – two associated events are totally sequential
- repetition (*) – associated event occurs zero or more times
- selection (|) – nonexclusive or, e.g., a|b means $a + b + (a\&b)$
- permutation (,) – associated events occur in any order, e.g., a,b means $(a;b)+(b;a)+(a\&b)$

A simple example using only control events would be the following scenario which describes legal access to a file:

Open;(Read | Write);Close*

Paraphrased in English, the above scenario is:

> Invocation of the procedure *Open* is followed by an arbitrary number of invocations of the procedures *Read* and *Write* which is then followed by the invocation of the procedure *Close*.

A simple example using only data events would be the scenario that describe the sequencing of a traffic light tl-1.

```
([signal-color(tl-1, 'red)];
[signal-color(tl-1, 'yellow)];
[signal-color(tl-1, 'green)])*
```

DPEs also supports conditional events which allow the analyst to qualify a control event with a predicate. A condition event follows a control event and is enclosed in square brackets. It may do one of three things. Make a parameter visible to the rest of the scenario, e.g., car in *move[car]*. Initialize a local variable, e.g., tl-1 in *change-light-color[tl-1 = tl]*. Or bind all free variables during the evaluation of a predicate, e.g., car in *move[exists (speed) car-speed(car,speed) and speed > 0]*. Variables once bound or parameters one made visible remain bound for the duration of the scenario (i.e., single assignment). Subsequent references to a variable or parameter are treated as predicates. The example below illustrates this point.

Open[file];(Read[file] | Write[file]);Close[file]*

During execution of a program which manipulates a file, the invocation of the *Open* procedure binds *file* to the opened file. Subsequent references to *file* by the remaining control events require that the *file* parameter be bound to the same opened file. In this way, when multiple files are opened and manipulated, distinct scenarios will exist for each opened file. This refinement corrects the error in the simpler scenario above which would have incorrectly been satisfied by the following more specific scenario:

Open[file1 = file];Close[file <> file1].

Paraphrased in English, *open* some file and then *close* some other file.

6.5.1.2 Extensions to Data Path-Expressions

Thus far the description of DPEs has been consistent with [39]. This subsection will describe three extensions that ASC makes to DPEs. First consider the following informal scenario:

> An aircraft is moved from one location to the next by the procedure *move*. The *location-of* the aircraft is updated to a known *next-location*. Updates to *location-of* should be performed by the agent *move*. All other updates to *location-of* are errors (i.e., fail).

This scenario is formalized via the following extended DPE. The **bold** text indicates extensions to DPEs.

> *(***t1**::*move[aircraft]* **agent A1**;
> *([aircraft.location-of = aircraft.next-location* **as-of t1***]* **done-by A1** +
> *([aircraft.location-of <> aircraft.next-location* **as-of t1***/{fail})))**

The first extension is temporal labels. **t1** from above is a label for the time when the control event was satisfied. This then allows the following data event to formulate a predicate which contains a reference to the state when **t1** occurred. In this example *aircraft.next-location as-of t1* is an expression which asks for the *next-location* of an aircraft as of the beginning of the *move* operation. In the context of this scenario, one is asking if the aircraft's current location is the intended next location. If it is not the intended next location, the scenario will flag a fail condition.

The second extension is agent identification. Agent identification is important because one may care not just that the execution is in some state but also what agent caused this state to become true. In this example, the analyst wants to know that the aircraft was moved to the next location and that it was done-by the labeled agent **A1** which is the procedure *move*.

The third extension refines the semantics of a control event. It is not illustrated in the above example. The normal semantics of a control event is that it is satisfied at the invocation (or start) of a procedure. This extension allows the analyst to place a modifier before a control event which causes satisfaction to occur at other times during the execution of a control event. The modifiers are "start", "complete", "not start", and "not complete". "Start" is the default modifier. The affect on the semantics of a control event is as the modifier name implies.

6.5.1.3 Scenarios and Anti-scenario as Validation Questions

Via DPEs the analyst is able to describe desirable and undesirable behaviors, known as scenarios and anti-scenarios, respectively. In the context of validating or debugging a specification they are known as validation questions. Validation questions are unique in that they may completely characterize some scenario or anti-scenario without requiring the specification to be complete. Figure 6.1 is an example of such a scenario. It will be referred to as validation question VQ1.

```
(([exits(p:position) track-position(T1, p) and
                    within(p, S1)];
  [within-handoff-computed-point(T1)];
  [within-accept-handoff-computed-point(T1)]) &
complete alert-controller[T1 = track]* &
complete handoff[T1 = track]) ;
[exits(p:position) track-position(T1, p) and
                   within(p, S2)]
```

Figure 6.1: A Successful Handoff Scenario for Validation Question VQ1

The English paraphrase of figure 6.1 is:

> When a track, T1, moves through several states beginning with being in the airspace S1, the event alert-controller may be completed zero or more times and the event handoff will be completed before the track, T1, enters the new airspace, S2.

The analyst has asked this question because he/she would like to validate that the stereotypical case of a track moving into a new airspace is properly handled, i.e., a successful handoff occurs.

The analyst refines this scenario by defining what a successful handoff scenario is: xxx

Alternatively, the analyst could have described the above situation as an anti-scenario, i.e., a description which describes a behavior the analyst does not want to occur. In this case, the anti-scenario is a track *T1* moving to the airspace *S2* while *handoff* has not yet completed (see figure 6.2.

```
(([exits(p:position) track-position(T1, p) and
                    within(p, S1)];
  [within-handoff-computed-point(T1)];
  [within-accept-handoff-computed-point(T1)]) &
complete alert-controller[T1 = track]* &
(not complete handoff[T1 = track]) &
[exits(p:position) track-position(T1, p) and
                   within(p, S2)])
```

Figure 6.2: A Handoff Anti-Scenario for Validation Question VQ2

Analysts use both scenarios and anti-scenarios because they are able to infer different results based on their success or failure during simulation. Success of a scenario tells the analyst that some path exists through the specification's behavior space. This raises the analysts confidence, but does not guarantee success in all situations. Failure of a scenario points out a problem with the specification and motivates its modification. Success of an anti-scenario has the same result as failure of a scenario in that it point out a bug. And finally, failure of an anti-scenario guarantees that this behavior is never possible. Finding this last result is often difficult since it requires traversal of the entire behavior space. In cases were the behavior space is infinite showing an anti-scenario to be unrealizable is impossible.

Later in this section, large or infinite behavior spaces will be used as one of the motivations for simplified specifications where the resulting behavior space is small enough that showing an anti-scenario to be unrealizable is possible.

The role a validation question plays during simulation of a specification is analogous to the role an input/output specification plays during testing an implementation. The task of the analyst differs from the task of the test engineer in that the latter has a well-defined, component or system which is believed to be complete and consistent upon which he/she run a test cases while the former does not have such expectations.

Determining success or failure of validation questions as described above requires that the specification be sufficiently complete so that the resulting behavior space is either

approximately correct and ready for validation or a superset of the intended behavior space and is thus amenable to pruning which focuses on subsections of the behavior space. A third approach is to make up for incompleteness by approximating what is missing.

Regardless of the approach, ASC supports the notion of focusing a specification based on the validation question. Such a focus will be identified by introducing a new system/environment boundary which partitions the specification based on the validation question. The validation question and associated concepts will be contained within the new system while remaining concepts will be contained in the new environment. The purpose of this partitioning is two-fold. One is to include in the system those concepts that are being tested during simulation with the expectation that these concepts are complete and at the right level of abstraction. And two to identify the other concepts as in the environment and thus may or may not be complete and are candidates for approximation.

The following subsection will describe how Influence Analysis provides the underlying reasoning capability needed to define this new system/environment boundary. Influence analysis will identify how one concept influences another and as a result highlight the dependencies between various concepts in the specification.

6.5.2 Influence Analysis

Brooks in [11] warns that descriptions of software that abstract away its complexity often abstract away its essence. Influence analysis is a means of allowing the analyst to see through this complexity to distinguish between concepts which are most relevant to the validation question and those which are not. Once identified, ASC provides the necessary tools to define a closed system made up of a simplified specification which defines a "system" and component approximations which define an "environment". The resulting simulation model will be amenable to simulation and analysis and adhere to Brooks' warning.

Influences finesse this issue by relying on rules which are easily computable and which generate all potential influences rather than making claims about actual influences. As such, the resulting influence graph should be considered a conservative representation of concept influences – that is they may indicate influences which are not actually possible, but are safe, in that they will not fail to indicate the presence of an influence that does exist.

Once the initial influence graph is generated, more knowledge intensive approaches are applied to remove many of those potential influences which are not actual influences. Some of these are done automatically, others require interaction with the analyst.

6.5.2.1 Influence Graph Definition

Influence analysis extracts a graph from a specification such that each declaration is a vertex and immediate influences between declarations are edges. There are three type of immediate influences: control, information, and VQ. These concepts are operationally formalized in terms of the ARIES Metamodel. Below is an intuitive definition of what these influences are.

- **Control influences** are concerned with influences which effect control flow during execution, i.e., *if* and *when* behaviors may occur. Some examples of this class of influence are references in the trigger of a demon, invocation of an event, and references in conditional statements.
- **Information influences** are concerned with the flow of information between concepts. Stated another way, when information changes, how does it percolate through the system? Some examples of these types of influences are: database updates, assignment statements, and definitional use of data declarations (i.e., relations, types, and instances) by other data declarations.
- **VQ influences** are concerned with influences on the validation question. These influences include all concepts referenced in the validation question's scenario.

Figure 6.3 is the Reusable Gist definition of the event *accept-handoff*. Figure 6.4 shows a paraphrase of this event. The resulting primitive influence graph is shown in figure 6.5.

```
Demon ACCEPT-HANDOFF(track,
                     current-controller:controller,
                     receiving-controller:controller)
    precondition controlled(track, current-controller) and
                 handoff-in-progress(track,
                                     current-controller,
                                     receiving-controller)
    postcondition controlled(track, receiving-controller)
    := steps(track.controlled ← receiving-controller;
             remove handoff-in-progress(track,
                                        current-controller,
                                        receiving-controller);
             track.track-status ← 'normal)
```

Figure 6.3: Reusable Gist definition of the event *Accept-Handoff*

The influence graph of figure 6.5 is most similar to de Kleer's mechanism graph from his work in qualitative reasoning [18, 19]. The mechanism graph shows the causal influences between concepts. A vertex contains an information value which represents a specific

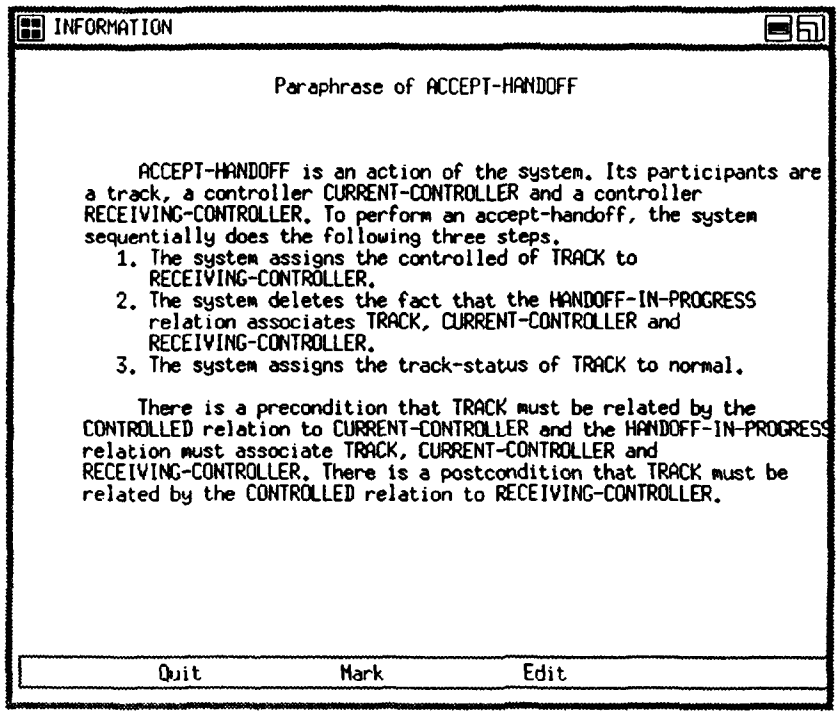

Figure 6.4: Paraphrase of the event *Accept-Handoff*

circuit component attribute (e.g. the voltage or current at a given component). Edges represent how a change in a vertex value is propagated to adjacent vertices. Edges are derived from either component models or domain specific heuristics. In influence graphs, a vertex represents a specification concept declaration (or fragment). Edges represent how a concept influences either the behavior or value of another concept.

The influence graph of figure 6.5 shows the immediate influences on and by *accept-handoff*. *Receiving-controller*, *current-controller*, and *track* are parameters of the event. *Enabling-pred-of-accept-handoff* is a composite node representing the precondition of the event. *Controlled*, *and*, and *handoff-in-progress* are relation referenced by the event. Edges within the graph represent the direction in which influences are propagated. Note that ***how*** each influence effects a given node is not represented. This is in fact outside the capability of influence analysis in ASC. None the less, this still provides a great deal of information to the analyst when creating a specialized specification as we will see later.

The goal of influence analysis is to focus on the influences in the specification which effect the validation question. This is accomplished by taking the transitive closure of these influences over the validation question to find all concepts which immediately or remotely influence it. The resulting set of influences is potentially quite large. One way to filter this set is to only include dynamic influences. The idea being that static influences do not change during the course of a simulation and thus do not effect its dynamic behavior.

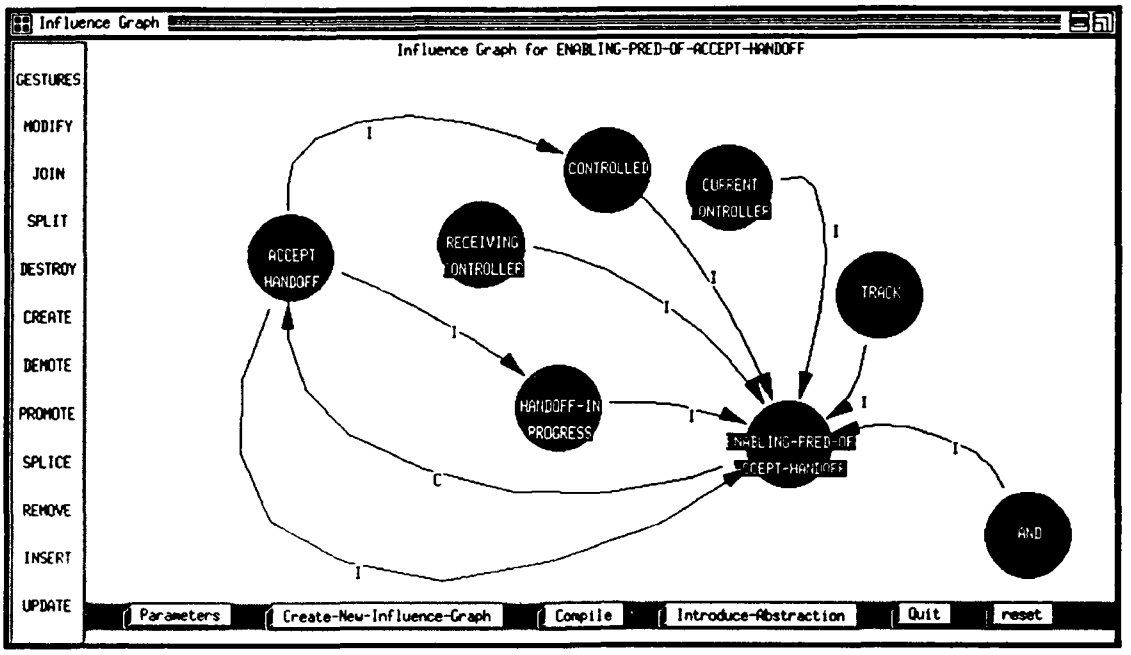

Figure 6.5: Primitive Influence Graph of the event *accept-handoff*

The remaining subsection address how to identify and abstract out these static influences based on information in the knowledge base.

6.5.2.2 Automated Graph Abstraction

Though the graph in figure 6.5 could be used as is, it shows many influences which really do not drive the dynamic behavior of the specification. This subsection will describe how some influence abstraction rules are applied automatically by ASC. Figure 6.6 shows the resulting influence graph. It is this graph, not the previous one, which the analyst is first shown after influence analysis.

Below are some of the abstractions which are automatically applied during influence analysis.

- Remove self referential influences.
- Remove influences from concepts known to not be "real influences" (i.e., many commonly used relations, e.g., *and*, are categorized in the knowledge-base as not having any dynamic influence).

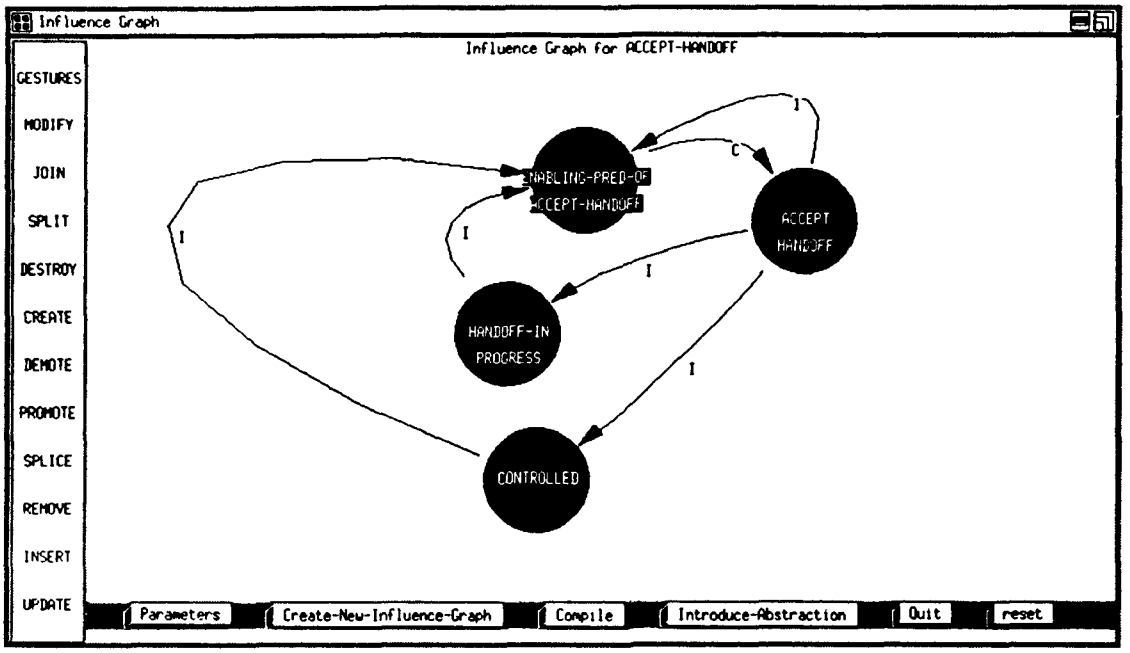

Figure 6.6: Influence Graph of the event *accept-handoff*

- Remove *static* concepts (concepts declared to not change during execution). If there is an influence on a static concept post it as an error instead.
- Remove parameters since they are dominated by the control influence on the event declaration they are a part of (i.e., who the caller is).

6.5.2.3 Interactive Graph Abstraction

Not all influence abstractions can be done in an automatic fashion. Typically, the presence of certain influences indicate either an error in the specification or an opportunity to apply an abstraction. The analyst must make these decisions. ASC identifies these cases during automatic influence abstraction and then posts notifications via an agenda mechanism. When the analyst is ready, he/she may view the agenda and the alternative actions recommended by ASC. Recommendations typically include suggested transformations which can cause the desired effect in either the evolving specialized specification or the underlying specification. Some of these interactive suggestions are:

- When there are no influences on a type or relation declaration, suggest that the concept should be declared static.

- When there is an influence on a type or relation declaration, suggest that the concept should be changed to dynamic (e.g., explicit or derived relation).
- When an influence node has only a single input information influence and only a single output information influence, suggest that the intermediate node be abstracted out and the input and output nodes be modified to be a direct influence.

6.5.3 A Validation Question and Specification's Behavior Space

This subsection will show how to partition a specification and then reformulate it into a simplified specification such that the simulation results of the simplified specification may be inferred about the original specification.

Consider the following partitioning of a specification:

$Spec = SYS + ENV$

where Spec is the original specification, SYS is the system side of the partition, and ENV is the environment side of the partition. SYS is defined to include the validation question and associated concepts (what concepts are "associated" will be defined later). The expectation is that these concepts are complete and ready for validation. Errors that result from this expectation not being correct are the ones simulation is intended to uncover. ENV is defined to include everything else. There are no expectations about its completeness.

Simulation of *Spec* does not prove anything. This is because ENV could have both extra and missing behaviors as well as its desired behaviors. These extra and missing behaviors affect the ENV behavior space in arbitrary manners and thus the overall behavior of *Spec*. If the analyst could ensure that some *Spec'* is a superset or subset of *Spec*, simulation results would be more meaningful. Consider the following cases:

- if scenario P is not realizable in $Spec^+$ then P must not be realizable in *Spec* and therefore there is an error in *Spec*.
- if anti-scenario P is realizable in $Spec^-$ then P must be realizable in *Spec* and therefore there is an error in *Spec*.

As a result of this observation, the analyst will approximate ENV to ensure that the resulting specification, to be known as the simplified specification, is either a subset or superset depending on whether the validation question is stated in terms of a scenario or an anti-scenario. More formally the simplified specification, SS, is defined as either:

$SS = SYS + ENV^+$ for scenarios

$SS = SYS + ENV^-$ for anti-scenarios

The next [subsubsections] will describe the partitioning operation followed by some extreme approximation operations based on generalization and specialization of the ENV partition to create either ENV^+ or ENV^-.

6.5.3.1 Defining a New System/Environment Partition

Defining a new system/environment partition begins with the validation question. Influence analysis identifies all concepts which are referenced by the validation question. All influence paths are then traversed back until they reach an active agent. All traversed concepts are included in the system SYS. All concepts which are influenced only by concepts already in SYS are added to SYS. All remaining concepts are part of the ENV partition.

6.5.4 Execution

The previous subsections described how to formulate the validation question and then how to reformulate the specification to create a closed simulation model. This final subsection will describe how the specialized specification is compiled and what happens during simulation.

The problem with simulating a high-level specification is that some of the constructs are not automatically compilable in the traditional sense. Particularly troublesome constructs in Reusable Gist are historical reference, control flow nondeterminism, and descriptive reference. Historical reference is handled during the reformulation phase. Transformations described by Feather in [23] are applied transform these references into appropriate record and access structures in the specialized specification. Descriptive reference is handled by taking advantage of that capability already present in AP5 [15] a lisp extension which the simulator uses extensively. And finally, control flow nondeterminism is handled at run time by the simulator. This subsection will describe the basic structure of the compiler and simulator followed by how control flow nondeterminism is realized in ASC.

The ASC compiler compiles most constructs directly into AP5, with the exception of events and invariants. Events are compiled into simulation tasks and invariants are compiled into automation rules. The state transition diagram associated with the main validation question is configured into the simulator so it will be animated during execution. The analyst may decide to configure other monitoring facilities. The driving scenarios are load into the simulator and the analyst initiates execution. The simulator then begins executing the driving scenarios. These in turn may initiate other tasks which are added to the simulation queue. The simulator manages multiple, parallel lines of control. When

there is no nondeterminism the simulator works pretty much like an generic discrete event simulator.

When the simulator hits a point during the execution where there is a choice on what it will do next, it has found a nondeterministic point. It could arbitrarily choose the next event and continue execution. In a completed specification this would be fine, but in a specification under development points of nondeterminism often are points where the specification has not been fleshed out fully. This is not to imply that the analyst must flesh these points out now if he/she is not ready to, rather ASC provides a variety of ways to handle these nondeterministic points. User interaction and probabilistic choice are established techniques which are often acceptable. Restrictive scenarios may also be used. Or finally, one may resort to arbitrary selection.

In all of the above cases, a wrong selection could be made. Violation of an invariant or scenario is often the first sign that this is the case. ASC automatically monitors these things and informs the analyst when they are violated. The analyst may then review the execution history to determine what happened. In some cases the violation is a result of an error in the specification, in others it is the result of an incorrect choice at a nondeterministic point. For the former, the analyst changes the specification. For the latter, the analyst may likewise change the specification by fleshing out the part of the specification responsible for making the choice. Alternatively, the analyst may on the fly define an additional scenario which guides the simulator through this point and allows execution to continue. The point of this facility is to allow the analyst to delay resolving this issue so that he/she does not get side-tracked from his/her current task.

Simulation continues until the driving scenarios are completed and the validation question is satisfied. At this point the analyst either has enough confidence in the current specification to continue with development or he/she may continue validation with the same validation question and a larger scope or he/she may validate other validation questions.

6.5.5 Related Works

One of the main problems with the testing approach to execution analysis is that it requires that the system description be relatively complete and consistent before testing can be performed, thus delaying execution analysis until later in the development process. The use of a test harness around individual modules mitigates this problem by allowing the analyst to test software modules independently from the rest of the system, but it still assumes completeness and consistency within the individual module. Simulation extends these capabilities by allowing the analyst to build simulation models which are abstractions of the system description. In such models, the analyst focuses on specific concerns while ignoring other aspects of the system. Common problems that need to be handled in order

to facilitate early execution are (1) undefined and partially defined terms, (2) inefficient or undefined algorithms, and (3) voluminous and therefore incomprehensible data. Below are some examples of KBSE systems which used simulation models in order to address some of the above problems.

The following capabilities are provided by knowledge-based simulation tools for software engineering: (1) an abstraction process which reduces the size of the behavior space to be validated, (2) the ability to approximate of incomplete portions of the specification, and (3) the ability to focus on selective views into the executing system. Additionally, simulations incorporate an execution environment which provide a suite of tools to facilitate client interaction during validation. Simulation still faces many of the same problems faced by the testing community—namely, selecting appropriate test cases and generalizing successful execution of test cases to broader claims about correctness of the specification.

6.5.5.1 Statemate

Statemate[36, 35], a CASE tool primarily built around state-charts, provides hierarchical abstraction and execution by allowing the analyst to run simulations at arbitrary levels of detail. The idea here is that, depending on what one is validating, different levels of detail are appropriate. Statemate allows the analyst to traverse the system state-chart decomposition and select which levels in the hierarchy will be active and thus drive the simulation. The main limitation is that the abstractions used during validation are the same ones built during construction. As a result, features which are orthogonal to the system decomposition are difficult to validate without running everything at the lowest level of detail and possibly being overwhelmed with simulation data.

- emphasis on execution at arbitrary levels in the abstraction hierarchy [hierarchy is the same for both validation and development]
- execution may be step-by-step, random , or exhaustive
- programmed execution – execution control language - sets break points, defines information (e.g., attributes to gather like time, memory, and other resource status) to gather.
- regression testing

This limitation of Statemate illustrates a general issue for simulation-based validation systems. That is, the ability to present information in a different notation from the one used during acquisition improves visual inspection. In a similar fashion, it is important in both visual inspection and simulation to be able to validate an abstraction of the system

that was not created when the specification was constructed. Such abstractions suppress details that are not relevant to the system feature being validated, making analysis of the abstracted specification more tractable and more focused.

6.5.5.2 PAISLey

PAISLey[81] addresses the issue of how to simulate incomplete specifications. If a function has not been specified, simulation is still possible. PAISLey supports either automatic selection of an arbitrary value from the function's range or inclusion of user defined approximations in the form of default values. The contribution here is that PAISLey augmented the simulation model so as to allow the analyst to perform validation on other aspects of the specification without being forced to define every aspect of the specification just to get executability.

6.5.5.3 Spec Critic

Fickas' group at U. Oregon has built several tools which directly address the problem of providing direction during specification validation. The Specification Critic [26] uses examples or scenarios to show how a specification may or may not satisfy higher level, informal requirements known as policies or goals. In this tool a user describes scenarios which illustrate good or bad sequences of events. The tool then runs the specification (an extended numerical Petri Net) to demonstrate that the scenario is possible or not. It is then up to the analyst to modify the specification to disallow the bad scenarios, but still allow the good ones. This tool was limited in that the links between specification components/scenarios and goals are hardwired.

ASAP [3] attempts to address this "hardwired" problem by using a planner to automatically find scenarios that show how specification components relate to client goals. In this work, goals are represented as allowed, prohibited, or desired states. These are provided to the planner as inputs of final states. Initial states also have to be provided by the user, since they can not be inferred by the planner. The net result of this work was a disciplined way to discover interesting test case scenarios and providing a way to run them against the specification. The scenarios served one of two purposes – as an existence proof (i.e., there exists some path through the specification which satisfies a given goal) or as a specification invalidator (i.e., there exists some path through the specification which violates a given goal).

6.5.5.4 Related Works in Safety

ASC is similar in approach to work in safety analysis which begins its analysis by defining safe and unsafe situations and then reasons from there to determine what parts of the system effect these safety condition. This methodology is described by Leveson in [52], "System safety analysis procedures often start by defining what is hazardous and then work backward to find all combinations of faults that produce the event." Toward this end safety analysts often select a weaker criterion of acceptable behavior to prove rather than attempting to prove the correctness of a program with respect to its original specification.

6.5.5.5 Partial Evaluation

Reformulation based on a validation question is analogous to how partial evaluation (mixed computation) [21] is able to generate an efficient residual program based on a more general program and a subset of its input parameters. This technique is potentially more powerful because a validation question is a richer source of knowledge than just a list of input parameters.

Not all optimizations are realizable during specialized specification construction. This problem is pointed out by Meyer in [56] when applying partial evaluation to imperative languages. The problem is that compile time execution can result in side-effects which are not noticed at the appropriate time. This is because the side-effect could happen during specialized specification construction and not during simulation. The problem with this is that other parts of the specification which trigger on the side-effects of the partially evaluated scenario will now not have those side-effects to react to at run-time. As a result partial evaluation at specialized specification construction time must be constrained not do anything that causes triggering states to disappear.

6.5.6 Evaluation of ASC

The success of the ARIES Simulation Component may be measured with respect to the following criteria.

- ability to execute a specification where previously it could not be done.
- ability to execute a specification with less effort than was required before.
- ability to document requirements satisfaction.
- ability to make validation comprehensible to stake-holders.

- ability to provide a flexible approach for system validation.

Toward these goals, ASC has made considerable progress.

Chapter 7

Evolution

Evolution is ubiquitous during requirements acquisition and specification construction. That this is so was recognized in the earlier phase of the overall KBSA project, and support for the evolution of formal specifications was built into the Specification Assistant. During the lifetime of the ARIES project, our belief in the need to provide support for evolution has continued, and a major part of the project has been directed toward supporting evolution in all phases of the now seamless activity of requirements acquisition and specification construction.

We begin by recapitulating the arguments for why evolution should occur; next we describe *evolution transformations*, the mechanism by which the earlier Specification Assistant provided support for evolution, retained and expanded in ARIES; we continue by describing the ways in which ARIES has gone beyond the Specification Assistant in its use of evolution transformations, namely by providing superior ways of *categorizing* evolution transformations (improving the user's ability to locate appropriate transformations, and understand their effects), and by integrating the transformations with the various user-oriented presentations (making presentations *two*-way, i.e., not only displays of information to the user, but manipulable objects through which the user can direct changes to the requirements/specification under construction). Finally, we examine how ARIES's other capabilities come into play as part of the above activities, and automated support for the addition of new transformations to the library.

7.1 Why evolution occurs

There are many reasons why evolution is a natural and unavoidable—indeed, frequent—activity in the course of requirements acquisition and specification construction. This was

recognized to hold for specifications at the start of the KBSA project; we continue to believe this to be so, and, in the course of merging specification construction with requirements acquisition, we see evolution to be pervasive throughout this combined activity. We outline our justifications for these beliefs below.

- *Incremental specification*

  Any formal specification of a large and complex system will itself be large and complex, in spite of the advantages provided by specification languages (provided by their freedoms from various implementation concerns). Thus it is unreasonable to expect that a formal specification can be constructed completely and correctly in one monstrous step (the so-called "big-bang" fallacy of specification construction). Instead, large specifications are best constructed and explained *incrementally,* in which each successive version evolves from its predecessor in some manner. It is important to realize that each stage of such an incremental development need *not* be constrained to be a pure refinement of the previous stage. As Balzer has pointed out [6], it is often useful useful to begin by telling the equivalent of "white lies"—incorrect but simplifying assumptions—and later retracting these, once the developer/reader has thoroughly understood the simpler version, and is ready for an *incremental* update. The common practice of first defining "normal case" behavior and then introducing exceptional cases is an instance of this principle.

- *Compromise of ideals*

  It is useful to allow the expression of, and reasoning with, *idealized* requirements, even if they are unattainable with present (or any imaginable) technology, are mutually contradictory, etc. This is so not only to permit incremental specification (as discussed in the previous item), but also to record those ideals (so that in the future we may respond to changes to take advantage of any new opportunities to better satisfy those ideals), and to make them explicit, and thus begin the process of *compromise* that will eventually lead to a consistent specification.

  It is in this process of compromise that *evolution* of idealized requirements takes place.

- *Response to feedback*

  The KBSA approach to software development encourages the early exploration of users' statements of requirements through the analysis and testing of the specifications that are constructed to realize those requirements. This feedback will lead to frequent evolution, i.e., change to those requirements, and the specification(s) that formalize them.

  Longer-term feedback will also continue to occur. While the KBSA approach is intended to improve the quality of the resulting code (especially in reducing the number

of changes made in response to the discovery of bugs in the implementation), it will not eliminate the demand for change once the constructed system has been fielded. It is inevitable that there will be aspects that the user will wish to change, be they modifications, extensions, or adaptations of the system to new and unanticipated situations. A crucial part of the KBSA paradigm is that such changes will *not* be made by hand-modification to the final program code, but rather by modifying the requirements or specification and *rederiving* the program code from the evolved specification. This is intended to make maintenance a speedier, and less error-prone activity. Balzer has observed that one consequence of this is that users will likely demand *more* changes, once they see the additional flexibility offered by this methodology! Thus evolution is going to remain a frequent activity, but will involve evolution of formalized requirements and specifications (the material manipulated by ARIES), not just the evolution of the final program code.

7.2 Evolution transformations—support for evolution

ARIES, like its predecessor the Specification Assistant, provides *evolution transformations* as the unit of support for evolution. The very purpose of evolution transformations is to elaborate and change specifications in specific ways. Like conventional "correctness-preserving" transformations (also called "meaning-preserving" transformations), they may be invoked by the user or by other transformations, and they are executed by a mechanical transformation system to cause changes to a specification. Correctness-preserving transformations are generally applied to derive efficient implementations from specifications, keeping the meaning of the specification unchanged; in contrast, our evolution transformations deliberately change the meaning of specifications. We do, in fact, include some meaning-preserving transformation in our library, but instead of using them for deriving efficient implementations, their purpose is either to reorder specifications (for better presentations), to rewrite specifications into equivalent forms using different language constructs, eliminate redundancies, or to make explicit some otherwise implicit features of the specification. Some of these transformations may also appear in a transformational implementation system, to be used to replace high-level specification constructs with low-level implementation ones.

The use of mechanized transformations provides the advantages of:

- *Mechanical assistance*—the burden of applying transformations is removed from the analyst, thus saving the analyst effort, and reducing the likelihood of error (when contrasted with the situation in which the analyst would have to conduct the changes by hand), and

- *Traceability*—the record of transformations provides traceability between the original form of requriements and their ultimate realization (perhaps in a compromised form) in the final specification.

These advantages were demonstrated in the development of specifications within the Specification Assistant. ARIES, in its role of supporting the seamless process of both requirements acquisition and specification construction, uses evolution transformations throughout. In the section that follows we examine how ARIES has addressed the problems of finding and applying evolution transformations.

7.3 Advances in ARIES with respect to evolution

7.3.1 Recap: the state of evolution transformations in the Specification Assistant

We began our exploration of evolution transformations by concentrating on two problems, a patient monitoring system and an air traffic control system, and worked out development scenarios by hand to discover what transformations were necessary. We then implemented general-purpose versions of those transformations, which could be applied to achieve those developments mechanically. The result of this exploration was a sizable library containing around 100 transformations, of a wide variety of types. This library formed part of the Specification Assistant. It was significantly more extensive than similar libraries developed by Balzer [5] and Fickas [25]. And while other researchers have studied evolution steps similar to those captured by our transformations [59, 42], they have not developed transformations to enact these steps.

7.3.2 ARIES advances

The ARIES system expands on the Specification Assistant work in several ways:

- *Improved coverage.* The transformation library of the Specification Assistant has been expanded so as to be applicable to a wide range of specifications.
- *Characterization of transformations.* The Specification Assistant organized its library of transformations into several coarsely defined groups. While this provided some help to the user in finding appropriate transformations from the library, it showed signs of trouble in scaling up to a larger library. In ARIES, we have worked to

characterize transformations in terms of their effects on several semantic dimensions. This is beneficial both to be able to assess the coverage provided by the library (i.e., to determine what range of transformations are required in the library in order for it to support a wide range of analyst activities), and to facilitate retrieval from the library (i.e., to find the appropriate transformation to make some desired specification change).

- *Broader range of applicability*

  Whereas the Specification Assistant operated only on specifications expressed in the specification language Gist [28], ARIES supports a wide spectrum of other notations, including hypertext, flow diagrams, state transition diagrams, and domain-specific notations. We needed a common internal representation capturing the semantics of all of these notations. By applying the transformations to the internal representation, the same transformation library can be applied to specifications expressed in a variety of different notations.

- *Retrieval of transformations through direct manipulation of presentations*

  The analyst views the requirements information accumulated within ARIES through *presentations*. ARIES's mechanisms make it possible for the analyst not only to *view* information this way, but also to *change* it. Through direct manipulation of the presented form of information, the analyst indicates to the system the nature of the change that he/she desires to conduct, and the system is then able to retrieve those evolution transformations that can or might achieve that change. Furthermore, ARIES organizes the retrieved transformations into a specialization hierarchy, so that if the application of one of the retrieved transformation would achieve a superset of the changes that another retrieved transformation would achieve, then the latter will be shown as a specialization of the former. The net result is to significantly ease the analyst's task of finding the right transformation to apply.

For the purposes of building and evolving the ARIES system itself, we have *avoided* hard-wiring manipulations at the presentation level to modifications to the underlying representation. Instead, we have followed a more staged approach that is far more flexible, permitting the rapid addition of new presentations and their associated modifications. The fundamentals of our approach are as follows:

Effect descriptions — characterizations of modifications in terms of their "effects" on each of several semantic dimensions. ARIES matches the effects of the desired modification against the effects achieved by the available evolution transformations in order to select the appropriate transformations.

Linking presentations to effects — each presentation suggests certain obvious and intuitive manipulations to the information being presented. These manipulations are linked to the corresponding effects, and these in turn are used to select transformations.

Linking evolution transformations to effects — each evolution transformation is characterized by the effects that it achieves. ARIES analyzes evolution transformations to determine some of their effects. This analysis is, when necessary, supplemented by information from ARIES developers.

The remaining sections of this chapter detail ARIES's mechanisms that make this possible: Section 7.4 describes the dimensions of semantic properties, how they are represented, and how changes are expressed in terms of them; Section 7.5 describes how the transformations are linked to effects (expressed in terms of changes to semantic dimensions), and how direct manipulations to presentations are linked to effects.

7.4 Semantic properties and effects

7.4.1 Dimensions of semantic properties

In our studies of specification evolution, we found the following dimensions of semantic properties to be important for characterizing the changes that occur:

- the modular organization of the specification, i.e., which concepts are components of which folders, and which folders inherit from which folders,
- the entity-relationship model defined in the specification, i.e., for each type what relations may hold for it, what attributes it can have, what generalizations and specializations are defined, and what instances are known,
- information flow links, indicating for each process or event what external information it accesses, what facts about the world it may change, and what values are computed and supplied,
- control flow links, indicating what process steps must follow a given process step and what process steps are substeps of a given process step,
- state description links, associating statements and events, on one hand, with preconditions and postconditions that must hold in the states before and after execution, respectively.

Each semantic dimension is modeled as an abstraction of the underlying representation, in the manner described in Section 3.5.1. Each abstraction consists of a collection of relations, each representing one aspect of the dimension described above. Thus, for example, the entity-relationship model is captured using the relations specialization-of, parameter-of, type-of, instance-of, and attribute-of. This model makes distinctions that are missing from many of the notations being supported. Thus E-R diagrams typically show specialization-of as just another relation in the application's data model. Here it treated not as part of the application's data model but as part of ARIES's language for structuring data models.

This semantic model captures information beyond what conventional notations typically show; however, conventional diagrams can be easily generalized to capture such information. For example, E-R diagrams are generally used only to show relationships among types, whereas our entity-relationship dimension also includes instances. Yet E-R-style diagrams could also be used to describe instances. The information flow dimension generalizes conventional data flow; it captures the flow of information that is not mediated by conventional message passing. Thus we can describe air traffic control as monitoring aircraft locations and changing them without implying that the aircraft are somehow sending location messages to the air trfaffic control system. Still, we could easily generalize conventional data flow diagrams to show such abstract information flow.

7.4.2 Generic network modification operations

Because we represent each semantic dimension as a semantic network of nodes and relations, we are able to identify a number of generic network modification operations which apply to any semantic network, and thus to each semantic dimension. The most primitive network manipulation operations are insert and remove for adding and deleting links, and create and destroy for creating and destroying objects. The meaning of an operation depends on the semantic dimension to which it is applied and the relation being affected; thus for example, the operation of adding a link in the information-flow dimension could mean making a process access information about an external object, whereas the same operation in the entity-relationship dimension could mean making one type become a specialization of another.

In addition to these primitive operations, we have identified a number of frequently recurring complex operations:

- update — remove a link from one node and add it to another node.
- promote — a specialization of update. If one of the linked nodes is part of an ordered lattice, then update the link so that it connects a higher node in the lattice.
- demote — the opposite of promote. Move the link to a lower node in the lattice.

- splice — remove a link from between two nodes A and B, and reroute the connection through a third node, C, so that A is linked to C and C is linked to B.
- split — replace a node A with two links B and C, linked together in some fashion, and where B and C divide between them the attributes of A.
- join — replace two nodes A and B with a node C, merging their attributes.

7.4.3 Examples of dimensions of semantic properties and changes within them

We sketch some instances of semantic properties that arise in our specification of air traffic control. The purpose of these examples is to show how information is actually captured along the different dimensions outlined above, and to illustrate the semantic distinctions that are made along each dimension.

- **modular organization**: the concepts of mass, direction, mobile-object, location are components of the physical-object folder. The concepts of aircraft, airport, control-tower etc. are components of the atc-model folder. Three folders are inherited-folders of the atc-system folder: 1) atc-model, containing objects and activities common to air traffic control, 2) system, containing definitions of various categories of systems, e.g., signal-processing-system, and 3) upper-model, a collection of generic concepts for modeling the semantics of natural language, defined by the PENMAN project [8]. The atc-model folder in turn has nine inherited-folders, including physical-objects, vehicle, system, and upper-model. The concepts in a folder may be defined in terms of concepts inherited from other folders, e.g., the atc-model's air-location is defined in terms of the physical-object's location.
- **entity-relationship model:** the specialization relationship is used to express the type hierarchy, e.g., aircraft is a specialization of vehicle which in turn is a specialization of mobile-object. Similarly, the instance-of relationship is used to express which types an object belongs to, e.g., the bartcc-facility is an instance-of the type atc-facility (BARTCC, Berlin Air Route Traffic Control Center, is the acronym for one of the air traffic control systems whose requirements we have been modeling).
- **information flow:** as was discussed earlier, information flow comprises the transfer of information (accesses to and modifications of information) between components, e.g., idealized versions of the ensure-on-course event access and modify aircraft locations, hence both kinds of information flow links, accesses-fact and modifies-fact hold between ensure-on-course and aircraft. Later on, some of these information flows are transformed into concrete data flows. The data-flow relationship expresses the

flow of data between components, e.g., from the radar process to the track-correlation function, and from the track-correlation function to the ensure-on-course process.

- **control flow:** there are two kinds of control flow links, control-substep and control-successor. The former captures the flow of control when an event consists of a series of steps; the relationship holds between the event and its substeps. For example, track-correlation has as a substep the operation to update an individual track. Control-successor holds between actions that are in temporal sequence: for example, ensure-on-course is activated whenever track-correlation updates tracks. A third category of control link, describing causal relationship between events, will need to be included as well, along the lines that Yue developed for the Specification Assistant [80].

- **state description:** links of this kind are between events and their pre- and post-conditions, e.g., a precondition to ensure-on-course taking action is that an aircraft be off course, and its postcondition is that the aircraft is back on course (at least in the early versions of the specification; in later versions its postcondition is that it has triggered the activity of notifying the controller, which ultimately will cause the aircraft to return to its course).

The meaning of a modification operation will depend on the semantic dimension to which it is applied.

- In the entity-relationship dimension, to insert a specialization-of link means to assert that one concept is a specialization of another, e.g., that the type surveillance-aircraft is a specialization of the type aircraft.

- In the information flow dimension to remove an accesses-fact link means to remove accesses to a category of external information from a component, e.g., to remove access by atc-system to the aircrafts' location-of relation.

- For the specialization links of the entity-relationship dimension, to splice means to assert that some type is intermediary to two other types in the specialization hierarchy, e.g., splicing military-aircraft between aircraft and surveillance aircraft.

- In the information flow dimension, to splice means to reroute an information flow between two components through an intermediary, e.g., splicing track between aircraft and the atc-system.

- In the control flow dimension, to splice means to re-route a direct control flow between two components through an intermediary, e.g., in the idealized versions of the specification there would be a direct control flow from an aircraft's maneuver process to ATC's ensure-on-course process, whereas in later versions this direct link would have been spliced through the track-correlation process.

Figure 7.1: Menu of modifications to the event-taxonomy presentation

7.5 Transformations, effects and semantic dimensions

7.5.1 Linking manipulations within presentations to effects

The modifications that are appropriate to each presentation are linked to the corresponding effects on the system's semantic dimensions, which in turn are used to select appropriate transformations. There are two ways that an analyst can indicate a desired effect for a given presentation. One method is to click on one of the gesture buttons on the left side of the presentation, labeled "MODIFY," "JOIN," "SPLIT," etc. The other mothod is to click right with the mouse on the object to be modified. Mousing on the object causes a menu of options to appear. The list of options is essentially the same as the list of gestures, except that the menu options indicate the types of nodes and links being operated on. For example, 7.5.1 shows the menu that comes up when one clicks on an object in an event taxonomy presentation. Observe that the menu reflects the underlying representation, as reflected in its stating the types of the nodes and links (**event-declaration** and **generalization**).

Once a desired effect is indicated, the presentation system constructs an effect description describing it. Effect descriptions consist of a generic modification operation together with a list of arguments. Arguments may be names of types and relations in the ARIES Metamodel, or specific instances of specification objects. For example, the effect description indicating that an event declaration is to be created consists of the operation **create** and one argument, the Metamodel type **event-declaration**. An attempt to delete the definition of handoff from a specification is indicated by an effect description consisting of the operation **destroy** and one argument, the event declaration named **handoff**.

The effect description is then passed to the transformation library retrieval mechanism to find all transformations matching the effect description. This retrieval mechanism attempts to classify the effect description against the effect descriptions of each transformation, to

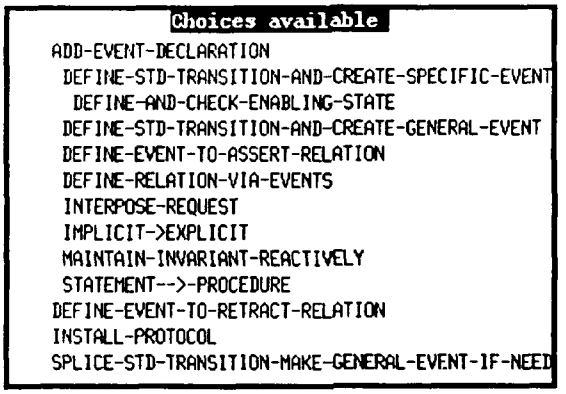

Figure 7.2: Menu of evolution transformations retrieved in response to selecting Create event-declaration node

see if the effects of the transformation subsume the effect indicated by the analyst. In order to subsume, the transformation must be guaranteed to achieve the desired result, when applied to the object(s) or type(s) indicated by the user. This determination must be performed on the specific objects being affected—it is sometimes the case that a transformation will not achieve the desired effect on any possible object that the presentation can display, but will achieve it on the specific objects the analyst selects. For example, transformations that operate on state transitions may be applicable to a given event declaration if the event declaration the analyst selects happens to be a state transition.

The product of the retrieval is a set of transformations which achieve the desired effect. Typically there is one transformation that matches the description most closely, and other transformations that are specializations of it. The set of transformations is presented to the analyst in a specialization hierarchy, so that the analyst can more readily see the relationships between the transformations. For example, selection of the Create event-declaration node item causes the system to produce the menu of evolution transformations shown in Figure 7.5.1.

The rationale for this approach is the assumption that often analysts will have a more complex modification in mind than a single edit to a single diagram. By showing the analyst more powerful transformations that match the description, the analyst may come across a transformation that more closely matches the true intended change.

The reliance on this subsumption-based approach makes it unnecessary to individually program menus of transformations for each presentation. Furthermore, the menus generated are more specific than any preprogrammed set of menus can be, because they represent the set of transformations that can be applied to the object selected by the analyst, rather than the set of transformations that can apply to any object of a given type.

7.5.2 Linking evolution transformations to effects

The principal effects of each transformation are explicitly recorded as part of the transformation definition. Each effect is a generic operation applied to a combination of the transformation's inputs, outputs, and other related objects which are not directly input or output. The transformation retrieval process matches each recorded effect against the desired effect indicated by the analyst in order to determine if the transformation is applicable.

In general, transformations can have two kinds of effects: *main* effects and *possible* effects. Main effects are guaranteed to result from transformation application (assuming that the goal of the transformation is not already satisfied). Possible effects may or may not result, depending upon the particular situation in which the transformation is applied.

ARIES analysis tools are able to partially analyze evolution transformations to determine their effects. This analysis is incomplete: is not intended to deduce *all* the possible effects of a transformation, but rather is intended to deduce particular classes of effects. These deductions are correct for typical transformations that we wish to encode. It is possible to imagine pathological cases where the deductions would be incorrect, but these cases do not occur in practice, so we have not been overly concerned with them.

The effect analysis process proceeds as follows. The analyzer scans the body of the transformation, looking for statements or expressions whose effects it can infer. For each of a variety of Common Lisp constructs, it has a rule for how the effects of the subexpressions of the construct contribute to the effects of the construct as a whole. For example, in the case of the Lisp progn expression the effect of the progn is taken to be the union of the effects of the individual statements in the progn statement. AP5 assertions and retractions on the knowledge base can be interpreted directly as insert and remove operations. If a transformation invokes another transformation, the effects of the invoked transformation are added to the effects of the invoking transformation. The analysis is performed repeatedly until there are no new effects added to any transformation.

Conditional constructs, such as if statements and loops, are handled differently. The then and else clauses of an if statement are interpreted as possibly executing if the surrounding if statement executes. Thus the main effects of the embedded clauses become possible effects. In a like manner, the effects of statements within loops are viewed as possible effects. The exception to this rule is if both the then clause and the else clause are seen to have effects in common. In that case, main effects are inferred for the if statement as a whole.

This approach has certain limitations, which make it an aid to transformation classification rather than an a fully automatic classifier. If the analyzer cannot recognize a construct, it cannot deduce effects for it. Also, the analyzer cannot deduce effects which are logically

implied by the observed effects. As a result, the effects that are automatically identifiable are a subset of the effects that are actually implied. However, in practice the great majority of effects can be inferred automatically.

A more serious potential problem can arise if the effects of subparts of a transformation interfere in some way. For example, if a transformation creates a specification component and then deletes it, the effect analyzer will fail to recognize that one effect undoes the other. However, this state of affairs is unlikely to occur in practice: well-formed transformations do not create structure and then destroy it. The simplifying assumptions in the effect analyzer about how transformation effects propagate are quite acceptable, although they would most certainly not hold for arbitrary Common Lisp programs.

ARIES also places its transformations into a specialization hierarchy, in which one transformation is more specialized than another if the set of changes that it causes is a superset of the set of changes the other causes. Again, this relies upon a combination of automated analysis supplemented by analyst-provided assertions to fill out this hierarchy. The developer asserts the specialization relationships, and the system verifies that all effects are subsumed. As was indicated earlier, the key to making this possible is the fact that ARIES represents information about itself, in particular, about its library of transformations.

7.6 Related work

The evolutionary approach to requirements specification has a number of precursors. Burstall and Goguen argued that complex specifications should be put together from simple ones, and developed their language CLEAR to provide a mathematical foundation for this construction process [12]. They recognized that the construction process itself has structure, employs a number of repeatedly used operations, and is worthy of explicit formalization and support—a position that we agree with wholeheartedly.

Goldman observed that natural language descriptions of complex tasks often incorporate an evolutionary vein—the final description can be viewed as an elaboration of some simpler description, itself the elaboration of a yet simpler description, etc., back to some description deemed sufficiently simple to be comprehended from a non-evolutionary description [29]. He identified three "dimensions" of changes between successive descriptions: *structural*—concerning the amount of detail the specification reveals about each individual state of the process, *temporal*—concerning the amount of change between successive states revealed by the specification, and *coverage*—concerning the range of possible behaviors permitted by a specification. We were motivated by these observations about description to try to apply such an evolutionary approach to the *construction* of specifications.

Fickas suggested the application of an AI problem-solving approach to specification con-

struction [24]. Fundamental to his approach is the notion that the steps of the construction process can be viewed as the primitive operations of a more general problem-solving process, and are hence ultimately mechanizable. Continuing work in this direction is reported in [68] and [2].

Karen Huff has developed a software process modeling and planning system that is in some ways similar to ours [40]. Her GRAPPLE language for defining planning operators influenced our representation of evolution transformations. Conversely, her meta-operators applying to process plans were influenced by our work on evolution transformations.

Kelly and Nonnenmann's WATSON system [51] constructs formal specifications of telephone system behavior from informal scenarios expressed in natural language. Their system formalizes the scenarios and then attempts to incrementally generalize the scenario in order to produce a finite-state machine. Their system is able to assume significant initiative in the formalization process, because the domain of interest, telephony, is highly constrained, and because the programs being specified, call control features, are relatively small. Our work is concerned with larger, less constrained design problems, where greater analyst involvement is needed. It is also more aimed toward the construction of specific behaviors starting from more general requirements. Nevertheless, we have recognized for some time that acquisition from scenarios is a useful complement to the work we are doing, in highly constrained design situations [43]. We have so far employed scenarios mainly in support of validation and debugging of specifications, as discussed in Section 6.5.

The work on classification of evolution transformations according to effects is closely related to current efforts in applying classification reasoners to software engineering. Two systems that are the most advanced in this area are the LaSSIE[20] and Comet systems[55]. Like ARIES, each one attempts to classify software components based on partial descriptions of functionality. Of the two, Comet has the most advanced approach, and the one most similar to ARIES, in that it employs structured descriptions of behavior. These descriptions are more closely tied to the structural form of the code, however, instead of the overall effect of the code.

7.7 Examples of evolution transformations

Figure 7.3 is a summarization presentation of one evolution transformation. Each of the 180 transformations of the ARIES system contains this form of information. The information helps analysts in several ways. The concept description appears in the ARIES-Manager window, when the analyst moves the mouse over a window or menu item associated with the transformation. Specialization information relates this transformation to others. Input parameters and output parameters describe parameter types and provide informal docu-

mentation which can be displayed to the analyst establishing parameters for an invocation of the transformation. Main effects and possible effects are used to determine if a particular transformation will modify the visible part of the specification.

Since transformations are represented internally as event declarations, ARIES functionality that applies to event declarations can be applied to transformations as well. This can be crucial for ARIES developers requiring guidance in future development of the system and for end-users demanding accountability for ARIES-initiated actions. Figure 7.3 is one example of this. Likewise, hierarchies of transformations can be presented using the event taxonomy presentatoin, as in Figure 7.4. This figure shows specializations of the transformation **add-relation-with-parameters**, a transformation which defines a new relation with given parameters. A number of specializations are shown, including the transformations that define states in state transition diagrams. This makes sense because states are a kind of relation in the ARIES model.

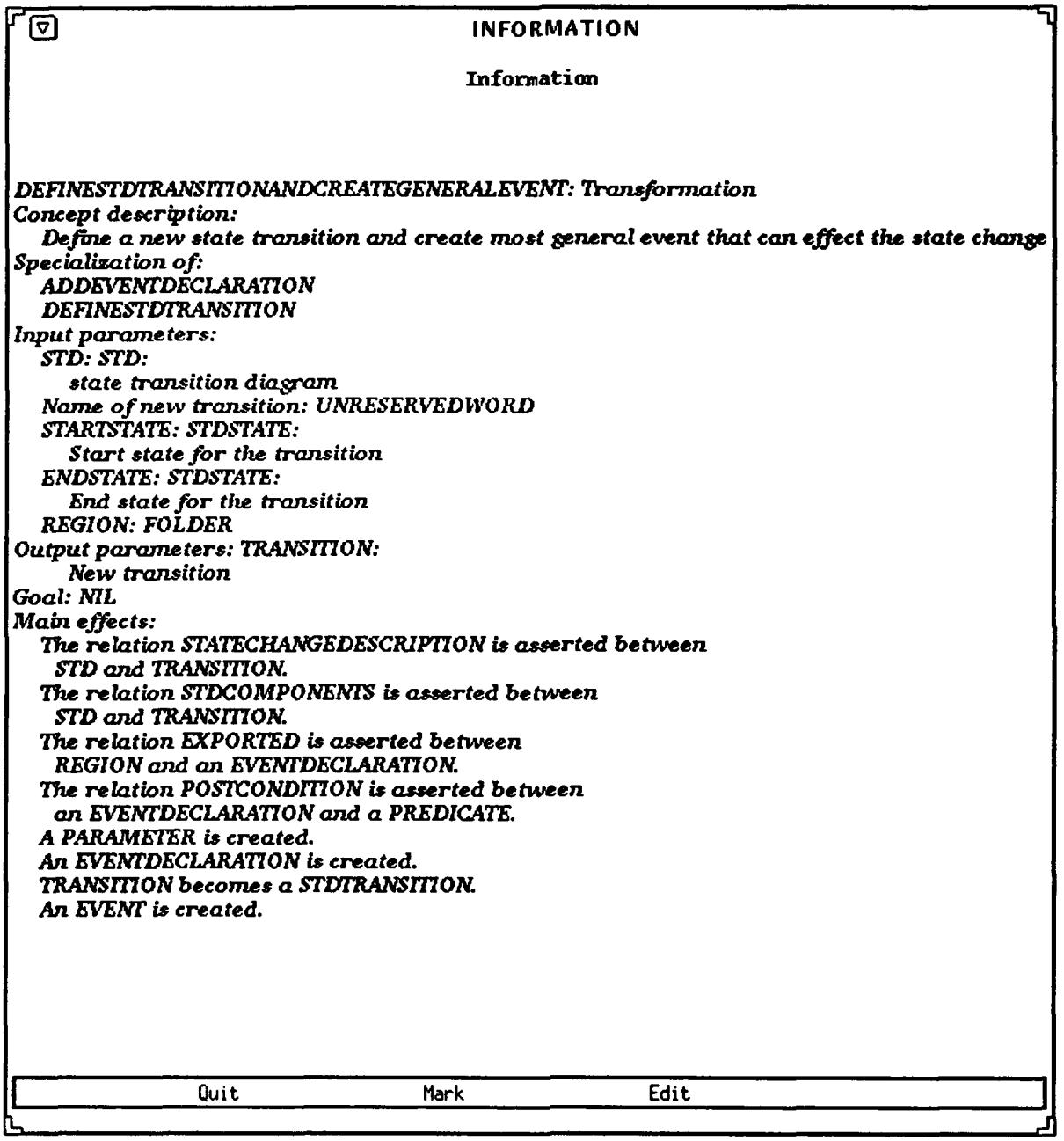

Figure 7.3: An example of an evolution transformation

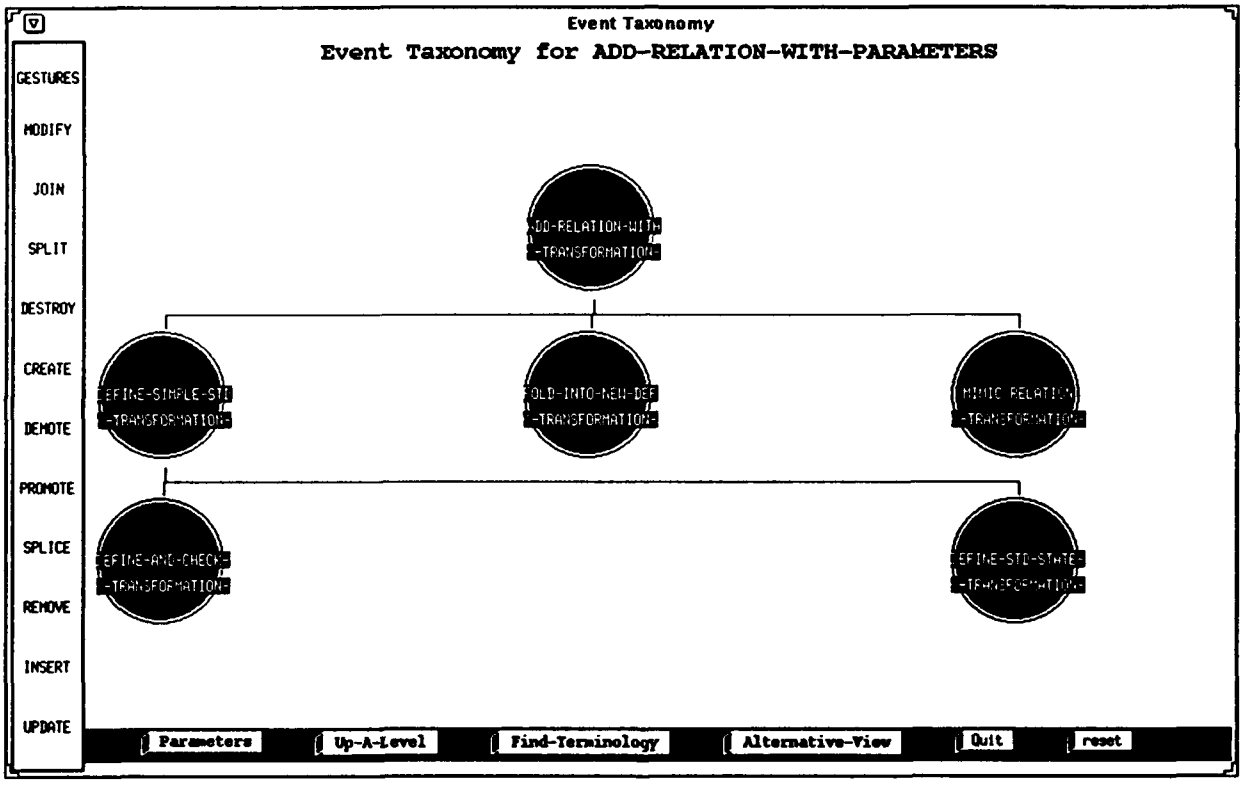

Figure 7.4: A taxonomy of transformations

Chapter 8

Future Directions

In this chapter we present several focal points for extending the work we have done on ARIES.

8.1 Improved acquisition and presentation modes

The current presentation system in ARIES is highly extensible. It is possible to introduce new presentations into the system will little effort, and we expect that more such presentations will need to be included. Feedback from potential users indicates that at the very least there needs to be more support for diagrams familiar to system engineers, such as conventional data flow diagrams and structure charts. Such extensions to not constitute major technical difficulties, but would greatly contribute to user acceptance.

A problem with the current system, however, is that it depends upon CLUE. The advantages of CLUE are that it is publically available at no cost, and it runs on a variety of hardware platforms. As a result, we were able to field versions of ARIES both on Lisp machines and on Unix workstations. The disadvantages of CLUE are that it is no longer supported by Texas Instruments, and it is written in Lisp. The use of Lisp for interface processing substantially degrades performance. Current interface builders are implemented in C or C++ instead of Lisp. In order to achieve acceptable performance for the presentation system, some reimplementation in C++ is likely to be necessary.

The ARIES presentation system meets the functional goals originally planned for it. Now that we have this functionality in hand, though, we can now identify new capabilities that would be desirable. These are outlined below.

8.1.1 Domain-specific acquisition and presentation

The presentations developed in ARIES up to now are all domain-independent. We would like in addition to be able to make use of domain-specific presentations for requirements acquisition. For example, we have looked at presentations of traffic-lanes, moving vehicles, and traffic lights as a means of capturing road traffic control specification. Depictions of air spaces are useful as background for describing air traffic control requirements. Lorna Zorman at ISI has made some initial progress in investigating the use of domain-specific presentations for requirements acquisition.

8.1.2 Acquisition using demonstration examples

Domain-specific presentations make requirements easier to understand, especially for domain experts. The use of examples or scenarios would further facilitate requirements acquisition. As described in Chapter 6, we already make use of scenarios to validate specifications. Acquisition via scenarios would also be useful as well. This is particularly important in the context of domain-specific presentations, as such presentations are most readily used as a medium for sketching out examples.

8.2 Representation issues

The representation framework in ARIES permits the definition of specialized abstractions, which in turn support the various presentations. Defining clean relational abstraction layers takes work, and not as many abstractions have been defined as would be useful. Some defined relations do not yet have add and delete methods defined, particularly when we have not yet determined what impact an update has on other relations. Information flow is one example of this. We are confident that many of such limitations can be gradually overcome in time.

An important step to take in this direction would be the development of a sublanguage for modular specification consistent with the underlying representations for types, relations, events, and invariants. While modularity is a strong component of many system analysis methodologies particularly useful in supporting teams of analysts, it will take some effort to map this notion of modularity onto specification level descriptions of environments and systems. We describe two cases in the paragraphs below.

8.2.1 Modularity for information flow diagrams

Event decomposition and event interdependency can be significantly different from functional decomposition and data flow. Recall that events have duration, possibly spanning multiple states in succession, and involving multiple entities of the system. Events can have preconditions, postconditions, and methods consisting of procedural steps. They may be explicitly activated by other events, or may occur spontaneously when their preconditions are met. They may have inputs and outputs. However, event definitions can affect the state of the system in ways other than generating outputs: they can assert and remove relations between objects, and create and destroy objects.

Our current abstractions capture four different notions of decomposition—part, physical component, logical component, and event. Part decomposition is simply a way of saying that the decomposition could be either physical or logical but either the analyst has not committed (in acquisition mode) or does not care (in review mode). Event decomposition is defined by the steps of the event body. When analysts state these steps explicitly (e.g., as in a recipe), ARIES can deduce the logical component view. However, if analysts do not make these steps explicit, it can be very difficult to automatically extract a decomposition. Analysts have difficulty in modularizing a monolithic description; hence ARIES will not easily accomplish this task automatically. Moreover, an analyst's way of viewing a decomposition of an event may not even come close to the notion of logical component of a system. The CYC ontology contains a very rich description of both spatial and temporal decomposition. From a temporal viewpoint, steps in an event are often smaller snapshots of the same event type and are not logical components in the system engineering sense. We advocate developing ARIES abstractions along the same lines. From these abstractions, ARIES would facilitate specification of many different kinds of event decomposition.

Even when analysts make logical decomposition strategies explicit, we still have problems in determining the module interconnections. In our model, analysts describe behavior through assertions which change participation in relations. For example, a move event results in a change to the location-of relation. It may not be appropriate to model location-of as data flow, or the location-of relation may be best associated with a specific submodule. We need to develop constraints which hold between a statement of information flow among modules and the collection of database assertions which describe the behavior of the modules. Automatic derivation of such constraints can be error-prone.

8.2.2 Modularity for state transition diagrams

The situation is much better for state transition abstractions, but there is still some work to be done. We need to extend our initial work on the propagation of change to the

specification when parts of the specification has been captured through non-state-transition formats.

Consider the case of an analyst modifying a state transition diagram of the enumerated relation **track-status**. This relation represents the various states of an aircraft with respect to an automatic tracking subsystem. The initial specification revolves around two statements:

```
type status := 'untracked, 'tracked;
relation track-status(aircraft,status)
```

If the analyst requests that "tracked" be split into "normal" and "coast," ARIES will invoke the **split-std-state** transformation which copies the old state and establishes the conditions for each new state to hold. This is equivalent to a specification with the following definition of status:

```
type status := 'untracked, 'normal, 'coast
```

On the other hand, if the analyst wishes to view the specification in more detail—opening up the "tracked" state—then a transformation is not called for. Rather, ARIES should change to a new presentation—a state transition diagram presentation of a new relation such as the **track-status-1** relation below.

```
type status := 'untracked, 'tracked;
type status1 := 'untracked, 'normal 'coast;
relation track-status-1(aircraft,status-1);
invariant ways-to-be-tracked forall(ac:aircraft)
    track-status(ac,tracked) iff
    track-status-1(ac,normal) or track-status-1(ac,coast)
```

Note, however, that ARIES can only derive the relationship between **track-status** and **track-status-1** (i.e., **track-status-1** captures a specific decomposition of a state-relation associated with track-status) by examining the invariant—leading to potentially intractable reasoning. One would prefer analysts to be able to state this decomposition explicitly, allowing ARIES to use this information to help focus its reasoning. When we put this decomposition feature into place, the ARIES prototype will deduce the intended state transition diagram from a request to open an existing state and be able to support decomposition of states.

8.3 Support for cooperation and reuse

ARIES goes further than previous systems in supporting cooperation within software projects, and supporting reuse across projects. Still, there is much work that remains to be done in this area. The following are some opportunities that the ARIES approach makes possible.

8.3.1 Merging workproducts and other CSCW support

Separate development of different requirements areas inevitably leads to inconsistencies. These inconsistencies are a natural consequence of allowing analysts to focus on different concerns individually. Although consistency is an important goal for the requirements process to achieve, we have concluded that it cannot be guaranteed and maintained throughout the requirements analysis process without forcing analysts to constantly compare their requirements descriptions against each other. Therefore, consistency must be achieved gradually, at an appropriate point in the specification development process. Nevertheless, it may not be possible to recognize all inconsistencies within a system description.

One technique that we have explored to facilitate reconciliation is the process of merging parallel elaborations [22]. Feather analyzed a restricted case of reconciliation, where different views of the specification are all derived by transformation from a common root specification, which describes the system in a very abstract way. His technique is to attempt to replay the various transformations in a linear sequence. By analyzing the transformations, their applicability conditions, and what they apply to, it is possible in many cases to determine automatically whether transformations applied to different views may interfere with each other.

The approach that we envision for ARIES centers of gradual elimination of differences between the conceptual models, regardless of their origin. If two members of an analysis team are using conflicting definitions of the same concept, they will each employ transformations step by step to eliminate those differences. In some cases this will involve having each analyst distribute the transformations that they employed so that the other analysts can employ them as well. As differences are resolved, specification components can be gradually promoted to the project-shared folders. In those cases where an analyst has employed a model that is more detailed than necessary for the shared model, abstraction transformations may be employed to reduce the detail to the level shared throughout the project.

8.3.2 Reuse constructions and retrieval

The folder mechanisms that we have built offer significant support in subdividing large specifications into manageable reusable pieces and segmenting knowledge into self-contained internally consistent units. Retrieval, understanding, and construction must rely on interconnections within the specifications themselves with little support at the folder level. Folders help manage the namespace but provide little guidance for actual specification construction via reusable components.

While we have concentrated mostly on the composition, understanding, and modification issues of reuse and have only rudimentary retrieval mechanisms in place, the underlying data structures could be used to support various deep retrieval approaches such as retrieval by reformulation (see [62], [63], [27]). In particular, the underlying representation for concepts serves as a foundation for automated classification—concepts in different folders are related to each other in a classification hierarchy.

Some additional structure at the folder level may also be helpful. We have observed that additional structure may be desirable in order to capture a folder developer's intentions for folder dependency. An approach taken by Larch provides an "assumes" link between specification fragments (referred to as traits in Larch). If a developer declares that a first trait is assumed by a second, then analysts know that the operators of the first will be additionally constrained in the second and that the full behavior is contained in both. Additional experimentation in the ARIES setting will be needed before committing to this type of folder interconnection.

8.3.3 Folder structuring and heterogeneous knowledge representations

One area that we have just started to explore is the use of folders to support the integration of heterogeneous representations. By "heterogeneous" we mean knowledge bases that were developed by different developers, either as part of different knowledge bases or as parts of the same knowledge base. In our work heterogeneous knowledge bases arise for two reasons. One reason is that we want to be able to draw on existing knowledge bases such as the Penman Upper Model, or existing knowledge bases developed in air traffic control, in order to lessen the work required to build a sizeable knowledge base. When we incorporated the Penman Upper Model into ARIES, we found inconsistencies between it and knowledge captured in other folders, that needed to be resolved. Another cause for heterogeneous knowledge representations is when multiple analysts develop models semi-independently, and then try to merge them back together. Such integration issues arise inevitably when developing large, reusable knowledge bases.

For example, one knowledge base may define physical-object as a natural kind with moving-object and stationary-object as subtypes. In a second knowledge base, entity is the uppermost concept. Simply declaring that physical-object is a kind of entity is an intrusive action on the first knowledge base—sound inferences may no longer apply. With a folder structure in place other less intrusive methods are possible. For example, in keeping with the package metaphor, we are interested in providing shadowing support which would allow a folder developer to create and use a new folder which shadows terms or even entire folders. In this shadowing folder, a physical-object can be redefined as an entity which includes all attributes of the original notion. When an analyst uses such a shadowing folder, physical-object specifications would inherit all the properties of entity and the original definition, yet there is no modification to the original knowledge base and other analysts can freely use it and have confidence in its guarantees.

This work should be seen as complementary to other work on hybrid knowledge representation, where cooperating decision procedures are defined between individual knowledge bases. Examples include [60], [50].

8.4 Additional intelligent assistance

ARIES plays a critiquing role in the specification evolution process. Constraint propagation mechanisms detect inconsistencies among nonfunctional properties. Evolution transformations have pre-conditions which are checked to see if the transformation can be applied and methods which propagate effects throughout the specification. There are many opportunities however for building stronger critiquing capabilities. Transformation pre-conditions can be strengthened and transformation effects can be increased to provide more substantial critiquing capabilities. We can illustrate the issue with one example requiring the interaction of transformations with domain-specific information. States in a state transition diagram should be mutually exclusive and collectively exhaustive (at least when a state transition diagram is in its final form). There are a number of stereotypical situations in which one state contains another (e.g., spatial inclusion comes up quite frequently in air traffic control specification) and transformations which check for these situations can help avoid errors (e.g., a transition which places an aircraft in an internal region without placing it in an enclosing region). Transformations which create states can look for potential anomalies such as violation of subregion relationships and can assert the necessary invariants which guarantee the exclusiveness of sibling states. We need to encode domain-specific constraints on typical state relations.

8.4.1 Formalism for the specification evolution process

Specification evolution can be viewed as a goal-directed planning process. Evolution transformations (and effect descriptions in particular) are a start at formalizing individual steps. but they do not formalize the planning aspects of the process. An enriched model including development goals and evolution plans expressed in a high level transformation formalism would greatly improve our ability to capture key design decisions, recognize evolution flaws, offer advice on subsequent development, and greatly assist our ability to manage a replay of a previous development.

8.4.2 Guidance for non-experts

As we indicated in Section 4.3, analysts who are inexperienced with a knowledge-based tool or its domain of application require more guidance than experts do. Current support comes in three principal areas. The Process Model presentation can be used to keep track of the current task being performed. The instructional mode provides guidance when working through examples. The transformation retrieval mechanism is used to suggest specialized transformations that the user may be unaware of when he or she suggests a modification.

There is much more that can and should be done in order to make a knowledge-based software engineering system such as ARIES easy for novices to use. Instead of simply guiding users through scripts, the instructional mode in ARIES should be able to guide the analyst in solving problems. The rudimentary plan recognition facility should be replaced with something capable of recognizing alternative plans. Ultimately, a deeper understanding will be needed of the process of requirements analysis in general and in the context of ARIES in particular. The capabilities of ARIES in the area of training support are merely suggestive of the kinds of support that will be needed in order for such powerful tools to become commonly used in practice.

8.5 Evaluation

The above discussion merely offers some suggestions of where further technology development might lead. However, the most immediate need at this point is not further technological development, but evaluation of the existing technology. Formative evaluations with potential users are indispensable for guiding future development. We need to find out what the limitations of the system are, and determine which of these limitations are fundamental ones. ARIES has been developed enough, and the ideas that it embodies have been tested extensively enough, that such evaluations can be expected to yield useful results.

Chapter 9

Acknowledgements

Sections of this report are adapted from previously published papers by the authors. Sections of Chapter 1 are adapted from Johnson, Feather, and Harris, "Integrating Domain Knowledge, Requirements, and Specifications," *Journal of Systems Integration* 1, Nov. ©1991, pp. 283-320, by permission of Kluwer Academic Publishers, Norwell, MA. Sections of Chapters 2, 3, and 4 are extended versions of Johnson, Feather, and Harris, "Representation and Presentation of Requirements Knowledge," submitted to *IEEE Transactions on Software Engineering.* Chapter 5 is adapted in part from Johnson and Harris, "Sharing and Reuse of Requirements Knowledge," *Proceedings of the 6th Annual Knowledge-Based Software Engineering Conference*, ©1991, pp. 57-66, by permission of IEEE Computer Society Press, Los Alamitos, CA. Parts of Chapter 7 are adapted from Johnson and Feather, "Using Evolution Transformations to Construct Specifications," in Lowry and McCartney, eds., *Automating Software Design,* ©1991, pp. 65-92, by permission of AAAI Press/The MIT Press.

Charles Rich has given helpful advice to this project. We would like to acknowledge current and previous members of the ARIES project: Jay Myers, K. Narayanaswamy, Lorna Zorman, Jay Runkel, and Paul Werkowski. This work was sponsored in part by the Air force Systems Command, Rome Air Development Center, under contracts F30602-85-C-0221 and F30602-89-C-0103. It was also sponsored in part by the Defense Advanced Research Projects Agency under contract no. NCC-2-520. Views and conclusions contained in this paper are the authors' and should not be interpreted as representing the official opinion or policy of the U.S. Government or any agency thereof.

Bibliography

[1] A.V. Aho and J.D. Ullman. *Principles of Compiler Design.* Addison Wesley, Reading, MA, 1978.

[2] J.S. Anderson and S. Fickas. A proposed perspective shift: Viewing specification design as a planning problem. In *Proceedings of the 5th International Workshop on Software Specification and Design, Pittsburgh, Pennsylvania*, pages 177–184. Computer Society Press of the IEEE, May 1989.

[3] J.S. Anderson and S. Fickas. Reasoning about user goals during specification design: Extending the planning paradigm. Technical Report CIS-TR-89-01, Univ. of Oregon, 1989.

[4] ASA. *Airman's Information Manual.* Aviation Supplies and Academics, Seattle, WA, 1989.

[5] R. Balzer. Automated enhancement of knowledge representations. In *Proceedings, 9th International Joint Conference on Artificial Intelligence*, August 1985.

[6] R. Balzer. A fifteen-year perspective on automatic programming. *IEEE Trans. on Software Engineering*, SE-11(11), 1985.

[7] R. Balzer, D. Cohen, M.S. Feather, N.M. Goldman, W. Swartout, and D.S. Wile. Operational specification as the basis for specification validation. In *Theory and practice of software technology*, pages 21–49. North-Holland, Amsterdam, 1983.

[8] J. Bateman. Upper modeling: Organizing knowledge for natural language processing. In *Proceedings of the 4th Intl. Nat. Lang. Generation Workshop*, Pittsburgh, PA, June 1990.

[9] H. Black. TTCP workshop on requirements engineering and rapid prototyping. Technical report, U.S. Army Communications-Electronics Command, 1989.

[10] A. Borgida, S. Greenspan, and J. Mylopoulos. Knowledge representation as the basis for requirements specifications. *IEEE Computer*, 18(4):82–91, 1985.

[11] F. P. Brooks. No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1986.

[12] R.M. Burstall and J. Goguen. Putting theories together to make specifications. In *Proceedings of the Fifth International Conference on Artificial Intelligence*, pages 1045–1058, August 1977.

[13] R.H. Campball and A.N. Haberman. The specifications of process synchronization by path expressions. In *Operating Systems*, pages 89–102. Springer-Verlag, 1974.

[14] D. Cohen. Symbolic execution of the Gist specification language. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 17–20. IJCAI, 1983.

[15] D. Cohen. *AP5 Manual.* USC-Information Sciences Institute, June 1989. Draft.

[16] P.R. Cohen and R. Kjeldsen. Information retrieval by constrained spreading activation in semantic networks. *Information Processing and Management*, 23:255–268, 1987.

[17] A.M. Davis. *Software Requirements Analysis and Specification.* Prentice Hall, Englewood Cliffs, N.J., 1990.

[18] J. de Kleer. How circuits work. In *Qualitative Reasoning About Physical Systems*, pages 205–280. MIT Press, 1986.

[19] J. de Kleer and J. S. Brown. Qualitative physics based on confluences. In *Qualitative Reasoning About Physical Systems*, pages 7–83. MIT Press, 1986.

[20] P. Devanbu, R.J. Brachman, P.G. Selfridge, and B.W. Ballard. LaSSIE: A knowledge-based software information system. *Communications of the ACM*, 34(5), 1991.

[21] A. P. Ershov. On mixed computation: Informal account of the strict and polyvariant computation shemes. In *NATO ASI Series, Vol. F14 - Control Flow and Data Flow: Concepts of Distributed Programming*, pages 107–120. Springer-Verlag, 1985.

[22] Martin S. Feather. Constructing specifications by combining parallel elaborations. *IEEE Transactions on Software Engineering*, 15(2):198–208, February 1989. Available as research report # RS-88-216 from ISI, 4676 Admiralty Way, Marina del Rey, CA 90292.

[23] M.S. Feather. Transformational implementation of historical reference. In B. Moeller, editor, *Constructing Programs from Specifications*, pages 225–242. North-Holland, 1991. Proceedings of the IFIP TC2/WG 2.1 Working Conference on Constructing Programs from Specifications, Pacific Grove, CA, USA, 13-16 May 1991.

[24] S. Fickas. A knowledge-based approach to specification acquisition and construction. Technical Report 86-1, CS Dept., University of Oregon, Eugene, 1986.

[25] S. Fickas. Automating the specification process. Technical Report CIS-TR-87-05, Department of Computer and Information Science, University of Oregon, 1987.

[26] S. Fickas and P. Nagarajan. Critiquing software specifications. *IEEE Software*, pages 37–47, November 1988.

[27] G. Fischer and H. Nieper-Lemke. Helgon: Extending the retrieval by reformulation paradigm. *Human Factors in Computing Systems CHI89*, pages 357–362, 1989.

[28] N. Goldman, D. Wile, M. Feather, and W.L. Johnson. Gist language description. Available from USC / ISI, 1988.

[29] N.M. Goldman. Three dimensions of design development. In *Proceedings, 3rd National Conference on Artificial Intelligence, Washington D.C.*, pages 130–133, August 1983.

[30] C. Green, D. Luckham, R. Balzer, T. Cheatham, and C. Rich. Report on a knowledge-based software assistant. In *Readings in Artificial Intelligence and Software Engineering*. Morgan Kaufmann, Los Altos, CA, 1986.

[31] B. Gruegge and P. Hibbard. Generalized path expressions: A high-level debugging mechanism. *Journal of Systems and Software*, pages 265–276, Dec 1983.

[32] R.V. Guha and D.B. Lenat. Cyc: A midterm report. *AI Magazine*, 11(3):32–59, 1991.

[33] R. Guindon. Knowledge exploited by experts during software system design. *Int. J. Man-Machine Studies*, 33:279–304, 1990.

[34] J. Hagelstein. Declarative approach to information system requirements. *Journal of Knowledge-Based Systems*, 1(4):211–220, September 1988.

[35] D. Harel. Biting the silver bullet: Toward a brighter future in system development. *IEEE Computer*, pages 8–20, Jan 1992.

[36] D. Harel and et al. Statemate: A working environment for the development of complex reactive systems. In *Proceedings, 10th International Conference on Software Engineering, Singapore*. Computer Society Press of the IEEE, April 1988.

[37] D. Harris and A. Czuchry. The Knowledge-Based Requirements Assistant. *IEEE Expert*, 3(4), 1988.

[38] D. R. Harris. A hybrid structured object and constraint representation language. *Proceedings of the Fifth National Conference on Artificial Intelligence*, 2, 1986.

[39] W. Hseush and G.E. Kaiser. Modeling concurrency in parallel debugging. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 11–20. ACM Press, March 1990. Published in *SIGPLAN Notices*, Vol. 25 No. 3, March 1990.

[40] K.E. Huff and V.R. Lesser. The GRAPPLE plan formalism. Technical Report 87-08, U. Mass. Department of Computer and Information Science, April 1987.

[41] V. Hunt and A. Zellweger. The FAA's Advanced Automation System: Strategies for future air traffic control systems. *IEEE Computer*, 20(2):19–32, February 1987.

[42] P. Johnson. Structural evolution in exploratory software development. In *Proceedings of the AAAI Spring Symposium on Software Engineering*, pages 35–39, 1989.

[43] W.L. Johnson. Specification via scenarios and views. In *Proceedings of the 3d International Software Process Workshop*, pages 61–63, Breckenridge, CO, 1986.

[44] W.L. Johnson. Deriving specifications from requirements. In *Proceedings of the 10th International Conference on Software Engineering*, pages 428–437, 1988.

[45] W.L. Johnson. Specification as formalizing and transforming domain knowledge. In *Proceedings of the AAAI Workshop on Automating Software Design*, pages 48–55, St. Paul, MN, 1988.

[46] W.L. Johnson and M.S. Feather. Using evolution transformations to construct specifications. In *Automating Software Design*, pages 65–92. AAAI Press, 1991.

[47] W.L. Johnson and K. Yue. An integrated specification development framework. Technical Report RS-88-215, USC / Information Sciences Institute, 1988.

[48] W.P. Jones. *'As We May Think?': Psychological Considerations in the Design of a Personal Filing System.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1988.

[49] Guttag J.V., J.J. Horning, and J.M. Wing. The Larch family of specification languages. *IEEE Software*, pages 24–36, September 1985.

[50] H.A. Kautz and P.B. Ladkin. Integrating metric and qualitative temporal reasoning. *Proc. Ninth National Conference on Artificial Intelligence*, pages 241–245, 1991.

[51] V.E. Kelly and U. Nonnenmann. Reducing the complexity of formal specification acquisition. In *Proceedings of the AAAI-88 Workshop on Automating Software Design*, pages 66–72, St. Paul, MN, 1988.

[52] N.G. Leveson. Software safety: Why, what, and how. *Computing Surveys*, 18(2):125–163, 1986.

[53] Julio Liete. *Viewpoint Resolution in Requirements Elicitation.* PhD thesis, Univ. of California, Irvine, 1988.

[54] M. Lubars and M. Harandi. Addressing software reuse through knowledge-based design. In *Software Reusability*, volume 2, pages 345–377. Addison Wesley, 1989.

[55] W. Mark, S. Tyler, J. McGuire, and J. Schlossberg. Commitment-based software development. To appear in *IEEE Transations on Software Engineering.*

[56] U. Meyer. Techniques for partial evaluation of imperative languages. In *Proceedings of Symposium on Partial Evaluation and Semantic-Based Program Manipulation*, pages 94–105, Yale Univ., New Haven, CT, June 1991.

[57] M.C. Mozer. Inductive information retrieval using parallel distributed computation. *Institute of Cognitive Science Report 8406*, 1984.

[58] J.J. Myers and W.L. Johnson. Towards specification explanation: Issues and lessons. In *Proceedings of the 3d Knowledge-Based Software Assistant Conference*, pages 251–269, Rome, NY, 1988.

[59] K. Narayanaswamy. Static analysis-based program evolution support in the Common Lisp Framework. In *Proceedings of the 10th International Software Engineering Conf.*, 1988.

[60] G. Nelson and D.C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. on Programming Languages and Systems*, 1(2), 1979.

[61] C. Niskier, T. Maibaum, and D. Schwabe. A look through PRISMA: Towards pluralistic knowledge-based environments for software specification acquisition. In *Proceedings, 5th International Workshop on Software Specification and Design, Pittsburgh, Pennsylvania, May*, pages 128–136. Computer Society Press of the IEEE, 1989.

[62] D. A. Norman and D. G. Bobrow. Descriptions: An intermediate stage in memory retrieval. *Cognitive Psychology*, pages 107–123, 1979.

[63] P.F. Patel-Schneider, R.J. Brachman, and H.J. Levesque. Argon: Knowledge representation meets information retrieval. *Proc. of First Conference on AI Applications*, pages 280–286, 1984.

[64] R. Prieto-Diaz and P. Freeman. Classifying software for reusability. *IEEE Software*, 5(1), January 1987.

[65] The KBSA Project. Knowledge-based specification assistant: Final report. Available from USC/Information Sciences Institute, 1988.

[66] S.P Reiss. Pecan: Program development systems that support multiple views. *IEEE Trans. on Software Engineering*, SE-11(3):276–285, March 1985.

[67] H.B. Reubenstein and R.C. Waters. The Requirements Apprentice: An initial scenario. In *Proc. of the 5th International Workshop on Software Specification and Design*, pages 211–218, Pittsburgh, PA, May 1989. Computer Society Press of the IEEE.

[68] W.N. Robinson. Integrating multiple specificationss using domain goals. In *Proceedings, 5th International Workshop on Software Specification and Design, Pittsburgh, Pennsylvania, May*, pages 219–226. Computer Society Press of the IEEE, 1989.

[69] W.N. Robinson. Negotiation behavior during requirement specification. In *Proceedings of the 12th International Conference on Software Engineering*, pages 268–276, 1990.

[70] D.R. Smith. Automating the development of software. In *Proceedings of the 5th KBSA Conference*, Rome, NY, 1990. Data Analysis Center for Software.

[71] R. Stallman and Sussman G. J. Forward reasoning and dependency-directed backtracking. *Artificial Intelligence*, 9:135–196, 1977.

[72] T. A. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, 10(5):494–497, September 1984.

[73] G.L. Jr. Steele. The definition and implementation of a computer programming language. Technical Report 595, MIT Artificial Intelligence Laboratory, 1980.

[74] G.L. Jr. Steele. *Common Lisp: The Language (2d edition).* Digital Press, 1990.

[75] W. Swartout. Gist English generator. In *Proceedings of the National Conference on Artificial Intelligence*, pages 404–409, Pittsburgh, PA, 1982. AAAI.

[76] L. Terveen. Person-computer cooperation through collaborative manipulation. Technical Report ACT-AI-048-91, MCC, 1991.

[77] R. C. Waters. System validation via constraint modeling. Technical report, MIT", Number= "AI Memo No. 1020, 1988.

[78] D.S. Wile. Integrating syntaxes and their associated semantics. Available from the author at USC/Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90292, USA - wile@isi.edu.

[79] D.S. Wile. Organizing programming knowledge into syntax-directed experts. In *Proceedings, International Workshop on Advanced Programming Environments, Trondheim, Norway*, pages 551–565. Springer-Verlag, 1986.

[80] K. Yue. Representing first order logic-based specifications in petri-net-like graphs. In *Proceedings, 5th International Workshop on Software Specification and Design, Pittsburgh, Pennsylvania, May*, pages 291–293. Computer Society Press of the IEEE, 1989.

[81] P. Zave. An insider's evaluation of PAISLey. *IEEE Trans. Software Eng.*, SE-17:212–225, 1991.

*U.S. GOVERNMENT PRINTING OFFICE