A METHODOLOGY, SPECIFICATION LANGUAGE AND AUTOMATED SUPPORT ENVIRONMENT FOR COMPUTER-AIDED DESIGN SYSTEMS

Duane Worley

June 1986
CSD-860038

UNIVERSITY OF CALIFORNIA

Los Angeles

A Methodology, Specification Language, and Automated Support Environment for Computer-Aided Design Systems

A dissertation submitted in partial satisfaction of the requirements for the degree of Doctor of Philosophy in Engineering

by

Duane Robert Worley

1986

© Copyright by

Duane Robert Worley

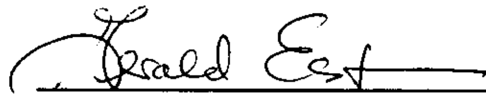
1986

The dissertation of Duane Robert Worley is approved.

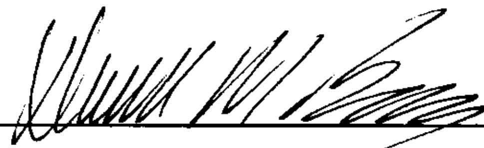
William Mitchell

Ismail Poonawala

Bertram Bussell



Gerald Estrin



Daniel Berry, Committee Chair

University of California, Los Angeles

1986

TABLE OF CONTENTS

page

1 Introduction 1
1.1 Motivation for the Research 1
1.2 Background 4
1.2.1 ISDF - Integrated Software Development Facility 4
1.2.2 SARA - Systems ARchitect's Apprentice 9
1.2.3 BIG CAESAR 11
1.2.4 Summary 13
1.3 Related Research Areas 14
1.4 Research Hypotheses 16
1.5 Dissertation Organization 17

2 Related Research 18
2.1 Related Research Projects 18
2.1.1 PALLADIO 18
2.1.2 GANDALF 19
2.1.3 COPE 21
2.1.4 USE 23
2.1.5 DEMETER 24
2.2 Related Research Areas 25
2.2.1 Software Engineering 25
2.2.1.1 Uses Relation 25
2.2.1.2 Information Hiding 26
2.2.1.3 Object-Oriented Design 27
2.2.2 Human-Computer Interaction 30
2.2.3 Human Problem Solving 31
2.2.4 Computer Aided Communication 33
2.2.5 Data Base Management 34
2.2.5.1 SYSTEM R - IBM 35
2.2.5.2 INGRES - UC Berkeley 35
2.2.6 Syntactic and Semantic Specification 36
2.2.6.1 Natural Language 37
2.2.6.2 BNF and Knuthian Semantics 37
2.2.6.3 LR(k) and S-attributes 38
2.2.6.4 LL(k) and L-attributes 39
2.2.6.5 Integral Help 39

3 The Design Methodology and System Organization 41
3.1 Introduction 41
3.2 Imprecise Goals of the Method and Kernel 42
3.3 The Design Method 43
3.3.1 The Conceptual Phase 44
3.3.2 Grammatical Expression of Dialogue 46
3.3.3 Logical Device Level Considerations 46
3.3.4 Physical Device Level 46
3.4 Overview of the System Organization 48
3.4.1 Conceptual Language and Support Software 51
3.4.2 Dialogue Language and Support Software 54

TABLE OF CONTENTS

page

1 Introduction 1
1.1 Motivation for the Research 1
1.2 Background 4
1.2.1 ISDF - Integrated Software Development Facility 4
1.2.2 SARA - Systems ARchitect's Apprentice 9
1.2.3 BIG CAESAR 11
1.2.4 Summary 13
1.3 Related Research Areas 14
1.4 Research Hypotheses 16
1.5 Dissertation Organization 17

2 Related Research 18
2.1 Related Research Projects 18
2.1.1 PALLADIO 18
2.1.2 GANDALF 19
2.1.3 COPE 21
2.1.4 USE 23
2.1.5 DEMETER 24
2.2 Related Research Areas 25
2.2.1 Software Engineering 25
2.2.1.1 Uses Relation 25
2.2.1.2 Information Hiding 26
2.2.1.3 Object-Oriented Design 27
2.2.2 Human-Computer Interaction 30
2.2.3 Human Problem Solving 31
2.2.4 Computer Aided Communication 33
2.2.5 Data Base Management 34
2.2.5.1 SYSTEM R - IBM 35
2.2.5.2 INGRES - UC Berkeley 35
2.2.6 Syntactic and Semantic Specification 36
2.2.6.1 Natural Language 37
2.2.6.2 BNF and Knuthian Semantics 37
2.2.6.3 LR(k) and S-attributes 38
2.2.6.4 LL(k) and L-attributes 39
2.2.6.5 Integral Help 39

3 The Design Methodology and System Organization 41
3.1 Introduction 41
3.2 Imprecise Goals of the Method and Kernel 42
3.3 The Design Method 43
3.3.1 The Conceptual Phase 44
3.3.2 Grammatical Expression of Dialogue 46
3.3.3 Logical Device Level Considerations 46
3.3.4 Physical Device Level 46
3.4 Overview of the System Organization 48
3.4.1 Conceptual Language and Support Software 51
3.4.2 Dialogue Language and Support Software 54

3.4.3 Device Languages and Support Software 56
3.4.4 Entity Relation Diagram 58
3.4.5 Auxiliary Support 61
3.4.6 System Librarian 63
3.5 Summary 64

4 Conceptual Definition Phase 65
4.1 Influence Diagrams 66
4.2 Object Identification Step 69
4.3 Relation Identification Step 72
4.3.1 Relation Types 72
4.3.2 Relation Specification 74
4.3.3 Relation Specification Conclusions 76
4.4 Attribute Identification Step 76
4.5 Operation Identification Step 77
4.5.1 Transformation Operations 77
4.5.2 Query Operations 79
4.5.3 Pre-transformation Validation Operations 80
4.5.4 Time-Invariant Validation Operations 81
4.5.5 Trade-offs 82
4.6 Conceptual Phase Compiler Outputs 83
4.6.1 Class Definitions 85
4.6.2 Operation Specification Output 92
4.6.3 Augmented ERD Output 94
4.7 Vis-a-Vis IFIP WG 2.7 Reference Model 94
4.8 Vis-a-Vis Ada Packages 97
4.9 Summary and Possible Improvements 97
4.10 The Concept Definition Language 99
4.11 The Influence Diagram Editor Specification 105

5 The Syntax Definition Phase 108
5.1 Vocabulary and Introductory Discussion 109
5.2 Syntax Specification 112
5.3 Augmenting Syntax with Semantic Actions 116
5.4 Grouping Terminal Symbols 117
5.5 Output Terminal Symbols 119
5.6 Aggregating Tool Syntaxes 119
5.7 Influence Diagram Editor Syntax 120
5.8 The Syntax Specification Compiler 123
5.8.1 Augmented Transition Network Output 125
5.8.2 Token List Output 133
5.9 Conclusions and Possible Improvements 134
5.9.1 Alternative Syntax Experiment 134
5.9.2 Why LL(1)? 135
5.9.3 Type Checking 137
5.9.4 Prompting and Menus 137
5.9.5 Imperative Sentences and Vocative Expressions 137
5.9.6 Simultaneous Input Sources 138
5.9.7 Multi-party Dialogues 138

6 The Logical and Physical Device Definition Phases 140
6.1 Introduction 141

3.4.3 Device Languages and Support Software 56
3.4.4 Entity Relation Diagram 58
3.4.5 Auxiliary Support 61
3.4.6 System Librarian 63
3.5 Summary 64

4 Conceptual Definition Phase 65
4.1 Influence Diagrams 66
4.2 Object Identification Step 69
4.3 Relation Identification Step 72
4.3.1 Relation Types 72
4.3.2 Relation Specification 74
4.3.3 Relation Specification Conclusions 76
4.4 Attribute Identification Step 76
4.5 Operation Identification Step 77
4.5.1 Transformation Operations 77
4.5.2 Query Operations 79
4.5.3 Pre-transformation Validation Operations 80
4.5.4 Time-Invariant Validation Operations 81
4.5.5 Trade-offs 82
4.6 Conceptual Phase Compiler Outputs 83
4.6.1 Class Definitions 85
4.6.2 Operation Specification Output 92
4.6.3 Augmented ERD Output 94
4.7 Vis-a-Vis IFIP WG 2.7 Reference Model 94
4.8 Vis-a-Vis Ada Packages 97
4.9 Summary and Possible Improvements 97
4.10 The Concept Definition Language 99
4.11 The Influence Diagram Editor Specification 105

5 The Syntax Definition Phase 108
5.1 Vocabulary and Introductory Discussion 109
5.2 Syntax Specification 112
5.3 Augmenting Syntax with Semantic Actions 116
5.4 Grouping Terminal Symbols 117
5.5 Output Terminal Symbols 119
5.6 Aggregating Tool Syntaxes 119
5.7 Influence Diagram Editor Syntax 120
5.8 The Syntax Specification Compiler 123
5.8.1 Augmented Transition Network Output 125
5.8.2 Token List Output 133
5.9 Conclusions and Possible Improvements 134
5.9.1 Alternative Syntax Experiment 134
5.9.2 Why LL(1)? 135
5.9.3 Type Checking 137
5.9.4 Prompting and Menus 137
5.9.5 Imperative Sentences and Vocative Expressions 137
5.9.6 Simultaneous Input Sources 138
5.9.7 Multi-party Dialogues 138

6 The Logical and Physical Device Definition Phases 140
6.1 Introduction 141

6.1.1 Interaction Tasks 141
6.1.2 Logical Devices 141
6.1.3 Physical Devices 142
6.1.4 Libraries 143
6.1.5 Why the Introduction of Logical Devices? 144
6.1.6 Run-time Scenario 146
6.2 Logical Device Definition Phase 146
6.2.1 Interaction Tasks 147
6.2.2 Logical Input 147
6.2.3 Logical Output 148
6.2.4 Logical/Physical Interface 149
6.3 Physical Device Definition Phase 150
6.3.1 Physical Input 151
6.3.2 Physical Output 151
6.3.2.1 Display Surfaces 151
6.3.2.2 Logical Screens on Display Surfaces 153
6.4 Interaction Environment 155
6.4.1 Tactile Alternation 156
6.4.2 Visual Alternation 157
6.4.3 Keystroke Errors 158
6.4.4 Appropriate Gesture 158
6.4.5 Physical Interaction Environment Specification 159
6.5 Device Libraries 163
6.5.1 Device Dependent Library (DDL) 163
6.5.2 Device-Independent Shape Library (DISL) 165
6.5.3 Logical Device Library (LDL) 166
6.6 Influence Diagram Editor Device Description 167
6.7 Complete Device Description Languages 169
6.8 Device Compiler 170
6.9 Summary and Conclusion 170

7 The SARA User Interface Management System 173
7.1 Introduction 174
7.1.1 Policy versus Mechanism 175
7.1.2 The Mechanisms 180
7.1.3 Chapter Organization 181
7.2 UIMS Characterization 182
7.2.1 Command-Response System 182
7.2.1.1 Work Space Memory 183
7.2.1.1.1 Instruction Memory 183
7.2.1.1.2 Environment Memory 187
7.2.1.2 Input Processor 190
7.2.1.3 Output Processor 190
7.2.1.4 Interaction Handler 191
7.2.2 State Representation System 193
7.2.2.1 Purpose 193
7.2.2.2 Introduction 194
7.2.2.3 Terminology 197
7.2.2.4 Computational Inverse Functions 200
7.2.2.5 Not-So-Instant Replay 201
7.2.2.6 Complete State Capture 202
7.2.2.7 Minimal State Capture 205

6.1.1 Interaction Tasks 141
6.1.2 Logical Devices 141
6.1.3 Physical Devices 142
6.1.4 Libraries 143
6.1.5 Why the Introduction of Logical Devices? 144
6.1.6 Run-time Scenario 146
6.2 Logical Device Definition Phase 146
6.2.1 Interaction Tasks 147
6.2.2 Logical Input 147
6.2.3 Logical Output 148
6.2.4 Logical/Physical Interface 149
6.3 Physical Device Definition Phase 150
6.3.1 Physical Input 151
6.3.2 Physical Output 151
6.3.2.1 Display Surfaces 151
6.3.2.2 Logical Screens on Display Surfaces 153
6.4 Interaction Environment 155
6.4.1 Tactile Alternation 156
6.4.2 Visual Alternation 157
6.4.3 Keystroke Errors 158
6.4.4 Appropriate Gesture 158
6.4.5 Physical Interaction Environment Specification 159
6.5 Device Libraries 163
6.5.1 Device Dependent Library (DDL) 163
6.5.2 Device-Independent Shape Library (DISL) 165
6.5.3 Logical Device Library (LDL) 166
6.6 Influence Diagram Editor Device Description 167
6.7 Complete Device Description Languages 169
6.8 Device Compiler 170
6.9 Summary and Conclusion 170

7 The SARA User Interface Management System 173
7.1 Introduction 174
7.1.1 Policy versus Mechanism 175
7.1.2 The Mechanisms 180
7.1.3 Chapter Organization 181
7.2 UIMS Characterization 182
7.2.1 Command-Response System 182
7.2.1.1 Work Space Memory 183
7.2.1.1.1 Instruction Memory 183
7.2.1.1.2 Environment Memory 187
7.2.1.2 Input Processor 190
7.2.1.3 Output Processor 190
7.2.1.4 Interaction Handler 191
7.2.2 State Representation System 193
7.2.2.1 Purpose 193
7.2.2.2 Introduction 194
7.2.2.3 Terminology 197
7.2.2.4 Computational Inverse Functions 200
7.2.2.5 Not-So-Instant Replay 201
7.2.2.6 Complete State Capture 202
7.2.2.7 Minimal State Capture 205

7.2.2.8 Comparison 205
7.2.2.9 The Mechanism 209
7.2.2.9.1 The Work Space Object 210
7.2.2.9.2 The Script Object 210
7.2.2.9.3 The Script Element Object 211
7.2.2.9.4 The Attribute Object 212
7.2.2.9.5 Walkthrough of the Do Operation 213
7.2.2.9.6 Mechanism Summary 214
7.2.3 Reference System 215
7.3 Conclusions and Suggestions for Further Research 216

8 SARA/IDEAS - A Computer-Based System Design Methodology 217
8.1 Design Procedure 218
8.2 Requirements Definition 221
8.2.1 Environment Assumptions 221
8.2.2 System Module 222
8.3 Structural Modeling 223
8.4 Behavioral Modeling 224
8.4.1 The Control Domain 225
8.4.2 The Data Domain 227
8.4.3 The Interpretation Domain 229
8.5 Building Block Library 229
8.6 Socket Attribute Modeling 230
8.7 Module Interconnect Description 231
8.8 Extensibility and User Interface 231
8.9 The SARA/IDEAS Environment 232
8.10 Summary and Conclusions 233

9 Conclusions 234
9.1 Contributions 236
9.2 Suggestions for Further Research 239

7.2.2.8 Comparison 205
7.2.2.9 The Mechanism 209
7.2.2.9.1 The Work Space Object 210
7.2.2.9.2 The Script Object 210
7.2.2.9.3 The Script Element Object 211
7.2.2.9.4 The Attribute Object 212
7.2.2.9.5 Walkthrough of the Do Operation 213
7.2.2.9.6 Mechanism Summary 214
7.2.3 Reference System 215
7.3 Conclusions and Suggestions for Further Research 216

8 SARA/IDEAS - A Computer-Based System Design Methodology 217
8.1 Design Procedure 218
8.2 Requirements Definition 221
8.2.1 Environment Assumptions 221
8.2.2 System Module 222
8.3 Structural Modeling 223
8.4 Behavioral Modeling 224
8.4.1 The Control Domain 225
8.4.2 The Data Domain 227
8.4.3 The Interpretation Domain 229
8.5 Building Block Library 229
8.6 Socket Attribute Modeling 230
8.7 Module Interconnect Description 231
8.8 Extensibility and User Interface 231
8.9 The SARA/IDEAS Environment 232
8.10 Summary and Conclusions 233

9 Conclusions 234
9.1 Contributions 236
9.2 Suggestions for Further Research 239

Figure 7.1: State Representation Mechanism 176

Figure 7.2: Circular History Queue 178

Figure 7.3: A Mechanism Template 180

Figure 7.4: Command-Response System 184

Figure 7.5: State Representation Template 198

Figure 7.6: Computational Inverse Functions 200

Figure 7.7: Not-So-Instant Replay 203

Figure 7.8: Complete State Capture 204

Figure 7.9: Minimal State Capture 206

Figure 7.10: Space and Time Comparison of State Capture Methods 207

Figure 8.1: The SARA Methodology 220

Figure 8.2: Structure Model of the Buffer System 224

Figure 8.3: GMB Control Graph for the Buffer System 226

Figure 8.4: GMB Data Graph for the Buffer System 228

LIST OF FIGURES

page

Figure 3.1: System Generation Data Flow 50
Figure 3.2: OReO Compiler 51
Figure 3.3: Grammar Compiler 55
Figure 3.4: Device Compiler 56
Figure 3.5: Data Base Compiler 58
Figure 3.6: Integration Facility 61
Figure 4.1: Forrester's Influence Diagram 67
Figure 4.2: Coyle's Influence Diagram 68
Figure 4.3: Worley's Influence Diagram 70
Figure 4.4: OReO Compiler 84
Figure 4.5: Influence Class Definition 87
Figure 4.6: Concept Class Definition 88
Figure 4.5: Entity Relation Diagram of Influence Diagram 95
Figure 5.1: OReO Compiler 124
Figure 5.2: Grammar Compiler 124
Figure 5.3: Noun Phrase ATN 126
Figure 5.4: Top Level Tool Dialogue ATN 127
Figure 5.5: Device Compiler 135
Figure 6.1: An Environment Alternative 145
Figure 6.2: Sample Screen Layout 154
Figure 6.3: Many Physical Devices Comprising One Display Surface 155
Figure 6.4: Interaction Environment 161
Figure 6.5: System Librarian 164
Figure 6.6: Device Compiler 171

VITA

October 28, 1949	Born, Los Angeles, California
1967-1971	United States Marine Corps
1978	A.B., University of California, Berkeley
1978-1982	Hughes Aircraft Company
1980	M.S.E.E., University of Southern California
1985	Engineer, University of California, Los Angeles
1985-1986	Rand Corporation

PUBLICATIONS AND PRESENTATIONS

Worley, D. R., (1985). *An Automated Support Environment for CAD System Development,* in the Proceedings of the 1985 ACM 13th Annual Computer Science Conference, New Orleans, LA.

Worley, D. R., Krell, E. A., (1985). *User Interface Specification in The SARA Design System,* in the Proceedings of IFIP WG 2.7 Working Conference held in Rome, Italy.

Worley, D. R., (1986). *The Role of State Representation in Interactive Systems.* Talk presented at the meeting of IFIP WG 2.7 at Princeton, New Jersey.

ACKNOWLEDGEMENTS

The members of the UCLA SARA/IDEAS Research Group: Daniel Berry, Bertram Bussell, Gerald Estrin, Rick Gillespie, Eduardo Krell, Dorothy Landis, Stuart Levin, Eddie Lor, and Dorab Patel, are acknowledged for their contributions.

Special thanks to Professor Gerald Estrin for providing a continuing research environment and for offering support and encouragement in all things academic and personal.

The work of Robert Fenchel and of Jim Foley's group at the George Washington University provided a substantial foundation for this work.

Thanks are also due to the members of IFIP WG 2.7 for their encouragement at a critical time in the evolution of the research. The many members of the Rand Strategy Assessment Center for tolerating my divided attention and for their support.

ABSTRACT OF THE DISSERTATION

A Methodology, Specification Language, and Automated Support Environment for Computer-Aided Design Systems

by

Duane Robert Worley

Doctor of Philosophy in Engineering
University of California, Los Angeles, 1986
Professor Daniel Berry, Chair

Computer based systems are becoming increasingly complex and expensive. Some have life critical responsibilities. There is a need for extensive design prior to embarking upon costly implementation phases. Many design tools and design techniques have been proposed and some built. Those tools that have been built are generally constructed to stand alone. They are, at best, loosely integrated with other tools. They share much commonality, notably the construction and manipulation of models of the complex system under design and a high degree of interactivity. A detailed study of areas of commonality guides the requirements definition of a Computer-Aided Design Of Computer Systems, CADOCS, support nucleus. The nucleus provides an integrated set of primitives. A specification language targeted to the support nucleus allows description of the design model and human-model interaction.

The current method of CADOCS system development is ad hoc. A new tool is developed from scratch or a great deal of effort is expended to integrate disparate tools into a system. It is the hypothesis of this dissertation that it is possible to define a methodology, a specification language, and an automated support environment suitable to the systematic development and execution of CADOCS systems. The tools of the UCLA SARA design system are selected as a test bed. The application of the methodology to the reimplementation of SARA and the resulting SARA/IDEAS system serves as an existence proof for the hypothesis.

The methodology is multi-phased and constructive. Following the methodology produces a specification that is compiled into an implementation of the specified CADOCS tool. All tools so constructed share many components, are cost-effectively built, are self-describing, and execute efficiently.

CHAPTER 1

Introduction

1.1 Motivation for the Research

The design of complex systems requires the combined talents and efforts of a cohesive group of designers. They need to rapidly explore design alternatives, communicate their views and results to their colleagues, arrive at decisions by consensus, and affect their decisions on the design. Computer Aided Design of Computer Systems, CADOCS, systems have been developed, for the most part, to satisfy the needs of the individual computer system designer. A CADOCS system comprises one or more software tools that allow the designer to create and exercise a model of the computer system under design. The current process of CADOCS system development produces nominally integrated systems that are difficult to use and that do not provide the benefits that true integration should offer.

CADOCS systems frequently begin as a single modeling tool. The development of the semantics of the model is ad hoc. Often, tool design proceeds based on an inaccurate or incomplete understanding of the model that the tool is intended to operate upon. As a requirements document, tool implementors are sometimes given a previously written user manual or the task of developing such a manual. This approach is intended to get the prospective tool user involved early in tool development. Many user manuals focus on the syntax of the tool rather than on the underlying semantic model. Tool syntax isn't necessarily the appropriate basis for design.

The design of the tool proceeds according to some accepted software engineering methodology, top down design [Jack83] or structured design [Your79] for example. Since tool development is time consuming, tools are often unintentionally obsolete before implementation is complete. While new products offer interaction devices with ever increasing sophistication, tool designers target to available devices [Oust81, Jens83], i.e. specific alpha-numeric display devices and keyboard input. Some tool designers incorporate general purpose graphic systems [Hanl79, Acqu82, ISO81] or develop only those capabilities needed by their tool.

This ad hoc approach shows up in other areas of CADOCS system implementation. Data base capabilities are often added on as an afterthought when the tool has proved its usefulness and users wish to store and retrieve their models. A general data base management system may be integrated by brute force or the host computer's file system may be used directly in lieu of a DBMS [Ciam76]. The data base provides a data representation of the model and fails to capture the semantics of the model [Hamm75]. Subsequent integration with another tool is almost obviated when the data base management system has no way to represent the semantic relationships that exist between two models.

Often, each tool provides its own interaction loop and input/output routines. Each must provide environment niceties like aliasing, command completion, spelling correction, command undo and redo, user help, prompting, menu generation, and accommodation of different user styles. The investment in a sophisticated user interface may far overshadow the cost of developing the single modeling capability and thus not appear to be cost effective.

The source of these problems may be that the tool developers are interested in the modeling capability and have an insufficient appreciation (budget or schedule) for human interface issues, data base management, and for the amplification achieved through integration of modeling tools.

The need to integrate CADOCS tools is inevitable and intense. Existing construction techniques produce tools with inconsistent syntactic interfaces, inconsistent use of DBMS, data bases without the semantic information necessary for integration, inflexible top down designed components not amenable to reusability, and inadequate multi-user support.

The complete integration of CADOCS tools to support collaborative design requires tools to present a consistent syntax and a common multi-user support environment, and a common data base that captures inter- and intra-model relationships.

Innovative design models and modeling tools have been, and continue to be, applied in the early phases of complex digital system definition [Estr78, Hill79, Alfo74, Boeh81]. In order for new modeling capabilities to achieve high payoff they must be inexpensively developed, be integrated with existing design environments, be understandable, and have high quality interfaces. A specification language to describe the proposed design model and the human-model interaction occurring in such systems, and a specification system capable of interpolating and analyzing that language should markedly improve the requirements analysis and design stages of tool development.

1.2 Background

Two representative CADOCS systems in use in industry and one representative CADOCS system from academia are compared and contrasted in this section. The focus of the survey will be on the historical development of the systems, but a discussion of functionality is also provided for perspective.

1.2.1 ISDF - Integrated Software Development Facility

The Integrated Software Development Facility, ISDF [Worl80, Wall81], is a CADOCS system synthesized from a collection of separately developed tools. ISDF is an evolving system in use at Hughes Aircraft Company. The component tools comprising ISDF are distributed across a variety of computer hosts including an AMDAHL 470, and a DEC PDP 11/70. It supports the development of real time software for embedded systems throughout the entire software life cycle [Hugh80].

Requirements Validation.

The customer's requirements must first be translated into a formal language, User Requirements Language, URL. URL is based on the Problem Statement Language [Tiec74], PSL, developed at the University of Michigan. URL is a language which provides the capability to describe a proposed system in a syntactically analyzable form. It is used in an interactive, computer-aided environment, and only fundamental information about the system needs to be stated in URL. The proposed system is described in terms of objects, properties, property values, and relationships. The language contains a number of types of objects and relationships which permit the following aspects of system requirements to be described:

system input/output flow,

system structure,

data structure,

data derivation,

system size and volume,

system dynamics,

system properties,

project management.

This URL description is stored in a data base that is accessible by the other components of ISDF.

Computer Aided Requirements Analysis, CARA, is based on both PSL and the Problem Statement Analyzer [Tiec74], PSA, which was also developed at the University of Michigan. Both PSL and PSA are results of an on-going project, Information System Design and Optimization System, ISDOS [Tiec74]. CARA is a software package which provides the ability to record URL system descriptions in a data base, to make modifications to that data base, to perform analysis, and produce various reports.

DAS/OFD [Will77] is the Operational Function Diagram Subsystem of the Design Analysis System. This software package interfaces with CARA, and provides not only an interactive interface, which facilitates the input of CARA data, but also provides a design analysis capability through the use of a general function model. The system definitions, OFDs, which describe major system functions, decisions, and interfaces, are input through a graphics interface. The general function model then automatically translates user inputs into an interactively controlled simulation of system operation providing performance and timing feedback. If the results indicate that

operational requirements have not been met, changes to specifications can be made and the specifications re-analyzed.

Design Verification

The Automated Interactive Design and Evaluation System [Jens83], AIDES, is a software tool developed at Hughes Aircraft Company. AIDES is an extension of the Structure Chart Graphics [Alli77], SCG, system also developed at Hughes. SCG provides an interactive graphics interface for automating design documentation standards.

AIDES is an interactive system which: collects design information, presents the design to the user in the form of structure charts and reports, answers questions about the design (goodness, testability, resource utilization, responsibility, etc.), and produces documentation for the customer. With guidance from the user, AIDES does some software design, tells the user how to test the system, tells the user the order in which to make the tests, tells the user the complexity of each test, and predicts the relative reliability of the software system. AIDES supports not only structured design, but also other software design techniques. AIDES collects and analyzes the software design structure or software architecture.

The Design Quality Metrics system [Haye81a, Haye81b], DQM, is an interactive, structured, design quality measurement tool. The system helps provide an independent and unbiased quantitative evaluation of the quality of a structure chart, where "quality" means adherence to structured design guidelines which tend to minimize problems during coding, testing, intergration, and maintenance. As such, DQM is a tool which can predict error-prone areas of a software design. Several metrics have been devised to determine system complexity and inter-modular cou-

pling, and have been validated against program errors experienced during program development.

The Distributed Data Processing Model [Will78, Jone81], DDPM, is a discrete event simulation model developed at Hughes that allows accurate, timely and inexpensive evaluation of distributed computer system design alternatives, (i.e., it is possible to evaluate such tradeoff alternatives as hardware selection, functional allocation to devices, operating system, and data base management system). DDPM embodies functions common to most systems and has been designed so as to be reconfigurable by specification of unique system variables. Typical outputs include device utilization, queue statistics, transaction occurrence and response times, and summary software execution results.

Reliable Program Production.

The Programmer's Workbench [Dolo76, Mash76a, Mash76b, Bian76], PWB, is a software package developed at Bell Laboratories. It is based on Bell Laboratories' UNIX time-sharing system, and runs on a DEC PDP 11/45, 11/70, or VAX 11/780. It provides a set of tools for software development. Remote Job Entry, RJE, provides for the submission and retrieval of jobs from an IBM or Amdahl host system (i.e. systems using HASP [IBM73] or JES2 [IBM75] interfaces), thus making possible the separation of development and execution environments. A Source Code Control System [Bona77], SCCS, provides facilities for controlling changes to files of text (e.g. source code, documentation). All versions of a file of text can be stored, updated, recorded, and retrieved, with updating privileges controlled. A Make System [Feld78] allows the specification of a dependency graph to automate the regeneration of a system of many interdependent modules (e.g. if changes are made to a module, affected modules will automatically be recompiled upon compilation of the

modified module).

The Complexity Path Analyzer [Stic77, Worl80], CPA, is a software tool developed at Hughes to run under PWB. It supports analysis of both the static and dynamic qualities of coded routines. CPA inputs syntactically correct code and produces both an instrumented source program (embedded with probes) and data which can be used to analyze the static characteristics of the code.

Auto-Test is another area under research and development at Hughes. Unfortunately, there are not yet enough software tools available which could ease the problems encountered during verification and validation testing. Much work remains in this area.

ISDF Critique

ISDF comprises a powerful set of analytic tools. The individual tools are written in dialects of FORTRAN, PASCAL, C, and BAL (IBM assembly language). Because of the method and level of integration, it fails to achieve the amplification that true integration could provide.

The ISDF user perceives no integration at the user interface front end. Each tool has its own user interface tightly integrated into the tool. This lack of consistency is mitigated by the fact that each phase of development, requirements, design, code, etc., are performed by different persons. Those tools having graphics interfaces have graphics software written specifically for that tool and targeted to specific display devices. There is no accommodation for other than keyboard input. None have integral help facilities.

Only marginal integration is perceived at the data base back end. Those tools that claim a data base generally use extended text file formats provided by the host computer's operating system. Integration of the various tools is attempted through the integration of their data bases into a common distributed data base. In some cases, the individual data bases are *integrated* by developing integrating software that is capable of handling the file formats of the various tools.

After all of the effort required to integrate the constituent tools, ISDF is in no better condition to accept the integration of yet another tool than before the integration. The construction of the system is not designed to insure that software components can be re-used and shared by other tools. In fact, the *system* is not designed at all; the individual tools are. There is no *kernel* that offers general services to all tools. Each tool supplies its own interactive loop, each its own graphics, each its own data base management system, etc. The only concession to designer collaboration is that the tools are re-entrant thus allowing simultaneous execution, and that the data base is protected from interfering access.

After presenting the design and integration plan of the ISDF system to Barry Boehm in 1980, he replied ''On [the proposal], my main advice is to be careful not to attempt to build a 'do-everything' tool box which is loosely integrated from large existing packages like CARA. We [TRW] tried this in 1975, and ended up with *an inefficient, hard-to-use kludge*'' (emphasis added).

1.2.2 SARA - Systems ARchitect's Apprentice

The SARA system has been designed at UCLA and implemented in PL/1 on the MULTICS system at MIT. SARA supports the design of complex concurrent digital systems. Both hardware and software design are supported. It provides tools

for the structural and the behavioral modeling of systems. Chapter 8 provides a detailed description of SARA and so only cursory characterization is given here.

SARA is composed of several tools that support digital computer system design from requirements definition to functional interpretation. Separate tools are provided for requirements definition, structural modeling, and behavioral modeling in the control, data, and interpretation domains. A building block library is defined to hold reusable components. It includes extensive on line documentation and an interface definition capability which supports integral help.

SARA Critique

SARA is a CADOCS system designed with extension considered. The interface specification subsystem anticipates and accommodates the inclusion of as-of-yet unknown tools. A detailed set of instructions are provided for a designer to define the user interface of a new tool. It provides a common and uniform interface to all tools as well as a uniform means of built-in help. The interface is textual only.

The Building Block Library provides a data base of previously verified models. The structure models and behavior models are defined textually and stored as text files. Although the building block library provides a SARA methodology specific data base, there is no general purpose data base.

A United Kingdom Task Force Study [Jack81] commented ''All of the SARA tools are written with a common command syntax, giving a particularly user-friendly uniformity to the whole system. There are global help facilities and documentation, which all component tools use, despite their having been developed by individual research students, and the current state of development can be determined by displaying the SARA tree – the structured hierarchy of SARA tools. All tools are interactive,

but as yet there are no graphical facilities.''

1.2.3 BIG CAESAR

The Big Caesar project [Katz83] attempts to integrate two apparently compatible VLSI design tools into a VLSI design environment. The first tool, Caesar [Oust81], is an interactive editor for VLSI layout. The second tool, the MIT Design Rule Checker [Bake80], DRC, is a design-rule/electrical-rule checker and switch simulator. Caesar outputs Caltech Intermediate Form [Mead80], CIF, as a description of the integrated circuit layout. Caesar, like many other VLSI layout tools, carries no information about design rules. A separate tool is required to perform design rule checking. DRC inputs CIF circuit descriptions, thus making it a logical candidate for integration with Caesar.

The original Caesar system allows a designer to interactively layout the geometry of an integrated circuit. Caesar was coded to interact specifically with an AED 512 graphics terminal [AED80]. Code dealt specifically with the fact that the terminal had 8 bit planes for color graphics, each layer of the chip is mapped onto a bit plane for display (not all AED terminals have all 8 planes of memory).

Katz [Katz83] states that ''The MIT design-rule checker works by converting a CIF description of a design into a rasterized image, based on a lambda grid. Templates are passed over the rasterization to check for violations; the display advanced by one lambda position, and the process is repeated. The checker reports the coordinates of the error.''

CAESAR Critique

The integration of these tools uncovered several problems. Rehosting Caesar into an environment including an AED 512 with only 6 bit planes lead the research group to replace the AED 512 with a Chromatics 7900. Modifying the interaction software to accommodate the new display device was a major activity. The previously advertised modeless interface was altered so that the user could invoke DRC.

Caesar does not support all CIF primitives, wires for example. The CIF output by Caesar was not entirely compatible with DRC. Integrating software was written so that DRC's error coordinates were not displayed as text, but were mapped back onto the display screen maintained by Caesar.

The integration of Caesar and DRC into a tool set called Big Caesar resulted in an environment stronger and richer than one containing just the two independent tools. Since neither tool was planned with later integration in mind, their merger resulted in a significant recoding activity. The user interface appears integrated because one tool has been made subservient to the other. The user interface integration was simplified by the fact that DRC was not designed as an interactive tool itself. Had it been, either one tool's interface would have to have been recoded to match the other, or the incompatible interfaces would have existed side by side.

No mention was made of error recovery, integral help, or any other user interface amenities. Data base management of designs was mentioned in passing and is clearly not a significant aspect of Big Caesar's design. Terminal independence was an add-on.

1.2.4 Summary

Many significant CADOCS systems exist today. Their designs and implementations have been evolutionary rather than planned. The result is cumbersome, difficult to use, expensive to build, and inconsistent design environments that fail to satisfy the needs of the design community.

Many claim to be *integrated* but an adequate definition of integration is lacking. Such a definition must include a consistent user interface provided by a common interaction handler, a consistent internal representation of models provided by a common near term memory manager, a consistent storage representation of models provided by a common long term memory manager, a consistent model of interactive execution provided by a common execution environment.

In addition to what *is* included in the common, tool independent portion of the CADOCS system, the definition must specify what *is not* included in the common portion. The variety of physical devices that must be accommodated and tool or methodology specific concepts must be excluded.

A test for extensibility of the CADOCS system must accompany the definition of integrated CADOCS systems. In other words, can another tool be added to the system without altering the common components previously identified?

Current systems support multi-user design by providing re-entrant code and by preventing concurrent write access to the shared design data base.

A successful, integrated CADOCS system must support design collaboration not only by prohibiting interference, but also by inhibiting duplication, by promoting the sharing of, review of, and consultation over, designs.

The previous requirements are imposed on CADOCS environments, regardless of the design space that they support. A separate definition of integrated CADOCS should be provided that is design space specific. This portion of the definition should focus on the tools' ability to support the entire life cycle of the design space. For example:

requirements,
structure,
behavior,
analysis,
simulation,
verification,
ancillary/support functions (documentation, etc).

1.3 Related Research Areas

There has been a large amount of diverse research conducted to improve the quality of design systems and of interactive systems. In general these projects have addressed a single aspect of the CADOCS construction and integration problem. Brought together, the results and techniques from these supporting areas can contribute a great deal to the solution of the problem at hand. Mechanisms for *human-machine interaction* are significant to interactive system design. Literature in this area concerns the tactile and visual responses of the user and the physical and logical interaction devices offered by the system. Familiar device examples include the mouse, keyboard, screen, and tablet. The physical device inventory requirements and user actions vary between batch and interactive systems and between systems presenting textual and graphic interfaces. These varying requirements must all be

considered. *Computer graphics* are discussed alone or as an element of human-machine interaction.

Although human-machine interaction is an important and obvious area for consideration by the CADOCS implementor, the CADOCS user requires a less physical view. An appropriate framework for discussion is the user's interaction with the design model under construction [Lill81], the *human-model interaction.*

A CADOCS system must provide an environment conducive to *human problem solving*. Work from the cognitive sciences can be applied to assure that the system supports problem solving rather than impedes it.

A method, that will lead the designer of a new CADOCS tool from original concept through to implementation, needs to be proposed. The method must be constructive, each step leaving the designer with a clearer picture of the new tool and closer to implementation. The field of *Software Engineering* offers insight into effective design methods. Arguments can be made for very high level specification languages [Balz83, Jaco83] and against them [Parn79]. Such a specification language may be provided that allows the designer to record a history of design decisions. The field of *formal languages* can contribute to the formulation and analysis of a specification language.

An *Automated Support Environment,* ASE, should provide an analytical capability to provide feedback to the designer at each step of CADOCS tool specification. The ASE must provide the capability to automatically generate significant portions of the CADOCS tool software. The ASE provides an integrated set of primitive functions that can be exploited by the designer of a new CADOCS tool. Considering the Uses Relation [Parn79] and practicing Information Hiding [Brit81] can contribute

towards the goal of reusability and abstract specification. Software Engineering has produced a great deal of literature about Design Environments [Solo84], particularly in relation to the ADA programming language [Fair80, DoD80].

1.4 Research Hypotheses

The primary hypothesis of this dissertation is that effective methods can be prescribed for specification, design, and construction of an environment for the design of complex systems. In order to test this hypothesis, it is necessary and sufficient to test the following list of secondary hypotheses.

1. A set of procedures can be defined to support the separation of user interface design from modeling software {see Chapter 4}.

2. A meta-language can be defined to express the conceptual model, syntax, semantics, and physical characterization of human-model interfaces for CADOCS systems. The meta-language can be computer processed by efficient means {see Chapters 4,5,6}.

3. A set of tools can be defined to analyze several important aspects of interface specification such as completeness and consistency {see Chapter 4,5,6}.

4. CADOCS tools have requirements for common resources. Those resources can be identified and codified into a kernel or nucleus support system sharable by any CADOCS tool {see Chapter 7}.

5. The method and support system can accommodate the addition of new tools into an environment of existing tools {see Chapter 9}.

6. A CADOCS system constructed around this method will achieve a substan-

tially higher level of integration than that achieved by present CADOCS systems {see Chapter 9}.

The SARA/IDEAS system currently under development will be a test bed for these hypotheses.

1.5 Dissertation Organization

After motivating the research and defining the problem a survey of related research is given in Chapter 2. The *meat* of the dissertation is in Chapters 3-7 where appropriate languages, tools, and procedures for specification of interaction in computer-aided design is described and exercised in one or more sample problems. The first two chapters will be devoted to describing the problems and issues in defining CADOCS systems, necessary attributes to be specified, and past and current research in specification and CADOCS construction techniques. Chapter 8 provides a review of SARA related research. Although specific conclusions and suggestions for further research are included in each of chapters 4, 5, 6, and 7, the dissertation concludes with a higher level discussion of the implications of the research.

CHAPTER 2

Related Research

Two types of related research are considered in this chapter, related research projects and related research areas. The former deal directly with the construction and composition of state of the art design environments. The latter offer results applicable to this problem domain but focus on only a single aspect of the problem.

2.1 Related Research Projects

2.1.1 PALLADIO

The PALLADIO [Brow83] system provides an integrated circuit design environment. The research project focuses on the need for integrated circuit design tools as opposed to self contained and isolated design aids. In addition to searching for the appropriate system features, PALLADIO researchers study the circuit design process and the suitability of tools and methodologies for that process.

The current implementation is written in INTERLISP-D running on a Xerox 1100. Interlisp-D augments Interlisp with bitmap graphics, among other things. Further power is added to the programming environment by LOOPS (Lisp Object-Oriented Programming System).

PALLADIO offers a fixed, menu driven user interface with an interactive graphics editors. A design library is mentioned, but an integrated data base is not. A general, event-driven simulator is provided that is capable of executing general circuit

behavior. Further research is being conducted to integrate expert-system aids into the PALLADIO environment. It is asserted that the expert-system can provide further automation in the effort of taking an abstract specification into a physical one.

The researchers conclude that the design process is only partially structured and multi-level.

2.1.2 GANDALF

Project Goals

The goal of the GANDALF Research Project [Habe79] is to create an environment for the collaborative development of large programs written in the Ada language.

''The GANDALF environment consists of three major components which together provide an integrated set of development support tools. The three components are

- a collection of Incremental Program Construction and Generation Tools,
- a collection of System Version Description and Generation Tools, and
- a collection of Project Management Tools.''

The Incremental Program Construction tool, IPC, is composed of a text editor, a compiler, and a dynamic debugger. No consideration of programming collaboration is given in the IPC. The syntax of Ada is central to all tools in the IPC. The mode of interaction is highly structured, i.e., each editing command takes a syntactically correct Ada program into another syntactically correct Ada program. Because of the tight coupling between tools within the IPC, it is easy to shift between editing source

programs and interactive debugging. In fact, source programs can be corrected during debug.

The System Version Description, SVD, portion of GANDALF uses the visible parts of Ada packages to describe the encapsulation or modularization of system design into system components. The interface specification of a system component provides sufficient information to potential users of that component without providing internal implementation or representation details. SVD enforces adherence to these specifications much like the IPC enforces adherence to Ada syntax. This capability eases the burden usually felt at system integration time when interface errors are traditionally found.

GANDALF's Project Management component inhibits the unauthorized interaction of programmers collaborating in a project. Modification of system components is restricted to specific users and a change history is automatically maintained.

Project Philosophy

"The GANDALF project favors a distinct view on how a group of programmers should collaborate on a software development project. This view holds that

- programmers share information about *system design* and *project state*, but hide *implementation details* from one another;

- a software development project in which several programmers collaborate is not merely a matter of *programming*, but also a matter of *organization* and *management*.

"The Ada language strongly supports this view by clearly separating the visible part of a package or task from its body. The visible part reflects the *design* of a system component; its body is its *implementation*."

The interface to GANDALF is dictated by the syntax of the Ada language. Some consideration is given to collaboration, but more to the prevention of interference. Mention of a data base is not specifically made, but an extended file system provides some of those capabilities.

2.1.3 COPE

COPE [Arch81a, Arch81b] is an integrated package of software providing an environment for cooperative program development. A full screen, syntax-directed editor, interactive execution supervisor, and a multi-purpose file system comprise the COPE system. COPE provides a specific textual user interface rather than a mechanism defining arbitrary interfaces. There is no indication that extension of the environment is planned for. Since syntax-directed editors are discussed elsewhere, only the interactive execution supervisor and file system are discussed here.

A series of interaction commands, some mapped to specific keys and invoked explicitly, others invoked implicitly, characterize the interactive execution supervisor. The interaction commands are **undo, redo, execute, resume**, **enter**, **submit**, and **quit**.

The interactive execution supervisor insures that each state changing command is recorded in the command log. A special window is provided on the COPE screen that shows the last command issued, i.e., the last command entered in the command log. Special function keys are provided for **undo** and **redo** capabilities. **undo** returns the system to its state prior to the last command invocation. Many commands

can be undone, each being moved from the *executed* portion of the log to the *pending execution* portion. The **redo** command moves a command from the *pending execution* portion of the log to the *executed* portion after re-executing it.

The COPE designers chose not to record all commands in the log. Cursor motion commands are examples of those excluded from the log process. The criteria for exclusion is that direct, easy to use inverses exist.

COPE provides for both the editing of and the execution of programs. The **execute** and **resume** commands take the system from the editing mode into program execution mode. Both commands are explicitly invoked, but have no function key provided. Editing commands and execution commands are treated uniformly with respect to the log file. Thus, COPE's **undo** and **redo** facilities are consistent throughout.

Log file entries are the result of the **enter** and **submit** commands. Both commands are invoked implicitly. Simple character operations (insert, delete, replace) are **enter**ed into the log and more substantial operations, (**execute, resume, quit**) are **submit**ted to the log.

The **quit** command, unlike the others, returns control to the host operating system. It is interesting to note that when the user re-activates COPE, the previous **quit** command is simply **undone**, returning the user to exactly the same state as before the **quit** command.

The file system is interesting to us here only in that it provides a common mechanism for storing the user's source program and for capturing the state change information including log file.

No treatment is given the data base aspects of the system. Extension of the system to include another tool requires additional programming of the user interface but the services of the interactive execution supervisor are available without additional cost.

2.1.4 USE

The User Software Engineering Methodology, USE, is described by Wasserman [Wass82, Wass84a] as a systematic means of constructing *interactive information systems*. The method is supported by a software tool set, the Unified Support Environment.

The method recognizes the importance of the user interface in interactive information systems, and supports user involvement early in the requirements analysis phase. Rather than a top-down design, USE produces an outside-in design. This method of specification separates the external interaction dialogue from the transactions on internal data representation. A major advantage of this technique is that many different interfaces can be developed to suit the needs of a specific user without altering the set of semantic operations or underlying data base schema.

Wasserman [Wass84b] provides the following list of goals for an interactive information system development method.

- functionality - The method should cover the entire development process, supporting creation of a working system that achieves a predefined set of requirements.
- reliability - The method should support the creation of reliable systems, so that users are not inconvenienced by system crashes, loss of data, or lack of availa-

bility.

- usability - The method should help the developer to assure, as early as possible, that the resulting system will be easy to learn and easy to use.

- evolvability - The method should encourage documentation and system structuring so that the resulting system is easily modifiable and able to accommodate changes in hardware operating environments and user needs.

- user involvement - The method should involve users *effectively* in the development process, particularly in its early stages.

- automated support - The method should be supported by automated tools that improve the productivity of software developers using the method; this requirement implies the availability of both a general set of automated aids and a method-specific set.

- reusability - The method should be reusable for a large class of projects and the design products from a given application should be reusable on similar future projects.

Finally, he states that USE, like other effective methods, prescribes a well-defined set of phases beginning with specification and analysis and terminating with a validated operational system.

2.1.5 DEMETER

Carnegie-Mellon University researchers have recently initiated the DEMETER project. It has many goals in common with this research, but no results have yet been published.

2.2 Related Research Areas

Researchers in the specific fields of software engineering, human-computer interaction, data base management, and formal language theory offer partial solutions that contribute to an understanding of the design and construction of highly interactive software systems.

2.2.1 Software Engineering

This dissertation defines a system that supports a broad user community and variety of CADOCS tools. The software engineering community offers several design techniques that offer useful suggestions.

2.2.1.1 Uses Relation

Parnas [Parn79] describes an approach to software design intended to result in systems that can be tailored to fit the needs of a broad variety of users. He identifies the *Uses Relation* that can be used to guide the design process towards that goal. The points most worthy of emphasis are described in the following paragraphs.

The requirements phase must consider subsets and extensions to the product under development. By identifying useful subsets of end user capabilities, flexibility becomes a design consideration rather than an afterthought. This approach not only produces a family of products that meets the needs of a wide variety of customers, but also provides a fail-safe way of handling schedule slippage.

The author defines the difference between software generality and software flexibility. "Software can be considered *general* if it can be used, *without change,* in a variety of situations. Software can be considered *flexible,* if it is *easily changed* to be used in a variety of situations." He further states that there appears to be an

inevitable run-time cost associated with a system exhibiting generality. By incurring a significant design-time cost designers can achieve flexible systems without incurring significant run-time cost. The extra design cost can be recouped if and when changes are required.

A prime goal of Parnas' methodology is the identification of the unit of change within a software system. Conventional programming techniques consider a module to be the unit of change, and define a module to be a subroutine that calls other subroutines. However, if one wants the modules to include all programs that must be designed together and changed together, then, one will usually include many small subprograms in a single module. The unit of change is not a single callable subprogram. The emphasis on subsets and extensions distinguishes Parnas' work from that of others who focus on hierarchically structured designs.

Finally, Parnas suggests that designing for subsets and extensions can reduce the need for support software such as separate SYSGEN programs, system specification languages, and special-purpose compilers. The price of the convenience features offered by such languages is often a compiler and run-time package distinctly larger than the system being built.

2.2.1.2 Information Hiding

One design concept from software engineering is information hiding [Brit81]. Put simply, information hiding produces an implementation that allows only controlled access to important data structures. Rather than making public the organization of data, a set of routines are made public that manipulate and access the data structure. The next section on object-oriented design describes in detail one variant of information hiding.

2.2.1.3 Object-Oriented Design – *Smalltalk-80*

In the early 1970s, at the Xerox Palo Alto Research Center, the Learning Research Group embarked upon the ambitious Smalltalk project. The proposed project had as its goal the design, test, and implementation of abstract notions conceived within the human mind and represented in computer hardware. The interface was to be natural and should require a minimum of translation from human thought to the computer representation of that thought. The Smalltalk project exemplifies object oriented systems and provides a vocabulary for further discourse. Fortunately, members of this group have published widely [Gold83, Gold84, Kras83].

Smalltalk-80 is an integrated programming environment. In addition to a programming language and an interactive graphics system it provides functions normally associated with an operating system. These functions include memory management, a file system, processor scheduling, display handling, and compilation.

The entire Smalltalk system is modeled on a form of object-oriented programming. At the highest level the user's interaction with the system can be viewed as an interaction between two *super-objects,* the User and Smalltalk. Delving deeper into Smalltalk reveals a multitude of smaller object types that serve a variety of roles. All data in Smalltalk are objects. Each object type can be further classified into more specialized subtypes. It is the communication and interaction between the various objects of the system that determine the functionality of Smalltalk.

The vocabulary of Smalltalk is quite small. The most important concepts are classes, objects, methods, and messages.

Objects, Methods, Classes, and Instances

In the Smalltalk programming language, the primary unit of data organization is the *object*. Everything in the system is represented and manipulated as an object. An object is a data structure consisting of local private memory and a set of operations, called *methods*, to manipulate information stored in the private memory or perform actions based on that information. The only way to access the private memory of an object is through the methods defined for that object. An object is similar to the concept of an abstract data type in other programming languages. For example, a stack abstract data type can be described in Smalltalk as an object consisting of:

Private Memory, the items on the stack.

Operations (Methods) on the private memory.
Push, Pop, Top, Height, Full, Empty.

Other examples of objects include numbers, character strings, queues, rectangles, file directories, compilers, text editors, and programs. A *class* is a description of a set of objects of the same type. The individual objects described by a class are its *instances*.

Messages

Smalltalk is a world of communicating objects. The methods of an object are its interface with the other objects in the system. The medium of communication between objects is the *message*. Whenever an object A (the *sender)* wants to get another object B (the *receiver)* to do something, A sends B a message which is interpreted by B's message interface. The message initiates execution of one of B's methods which calculates a response, and B then returns this response to the sender. This is the primary way of making things happen in Smalltalk.

Subclasses, Superclasses, and Inheritance

Another important concept of object-oriented programming in Smalltalk is the *subclass*. A subclass is a class of objects possessing all the variables and methods of some other class, called its *superclass*, except for certain explicitly stated additions that extend or override the variables and methods of the superclass.

A simple analogy to the subclass can be found in traditional dictionary definitions of English words. For example, a *man* can be defined as a male adult human. A *father* can be defined as a man who has one or more children. The class father is a subclass of the class man, and the class man is a superclass of the class father. Notice that father is defined in terms of man but is more specialized. In order for an object to be a father it must not only be a man but it must also have one or more children. All fathers are men but not all men are fathers. A subclass is thus a proper subset of its superclass.

Another idea important to subclasses is *inheritance*. A subclass is said to inherit all the attributes of its superclass. A father possesses all the attributes of a man, with some additional attributes not required of the man class. A Smalltalk subclass inherits all the variables and methods of its superclass but it also adds new variables and methods in its implementation description, or it may override a method of its superclass with a new definition. Inheritance is transitive. If class B is a superclass of class C and class A is a superclass of B then, by the definition of inheritance class, C inherits the attributes of class A is well as those of class B.

Summary

Smalltalk is an interactive programming environment with a graphical interface. Designing a Smalltalk program requires an implementation of each of the program's objects including a visualization of that object. To support the desired graphical interaction, Smalltalk expects a high-resolution graphical display screen and pointing device such as a mouse.

The Smalltalk environment provides many facilities often found in conventional operating systems, including: memory management, file system, graphical input and output device handling, debugger, performance monitor, compilation and decompilation.

2.2.2 Human-Computer Interaction

Human-computer interaction in early interactive software is characterized by textual command-response systems. The human user uses an alphanumeric keyboard to enter commands and the computer system gives a textual response on an alphanumeric display device. Current interactive operating systems still rely on textual command-response for human-computer interaction.

The textual command-response approach is dictated in part by the expense and paucity of devices to support graphical interaction and in part by the scarce specialized knowledge required to implement graphical interaction software. Advances in technology have provided economical physical devices to support graphical interaction and standardized software packages have been developed that allow the cost of graphical interaction software to be amortized across many applications.

Two well known software standards are the CORE graphic standard [Fole81, Acqu82, Hanl79] and the Graphic Kernel Standard, GKS [ISO81], A graphic package implementing either standard contains algorithms and data structures to support

graphical interaction and it defines a set of procedure call interfaces that comprise the implementation's interface to the using application program. The graphical representation of an object is maintained by the graphical package and the application program must maintain the semantic representation separately. This separation requires the application program to maintain bi-directional linkages between the two representations.

Foley and Van Dam [Fole82] describe human-computer interaction as a sequence of abstractions. The highest level of abstraction is the *interaction task* that the user carries out by using a logical device. A logical device is a software implementation that allows the user to carry out a specific interaction task on a specific physical device. The abstraction sequence is extended and integrated into the results of this work as described in Chapter 6.

2.2.3 Human Problem Solving

The *Psychology of Human-Computer Interaction* presents a Model Human User, MHU, that describes the physiological and psychological aspects of the human-computer interface [Newe72].

The MHU is characterized as a system and a set of principles of operation. The system is composed of three subsystems: the *Perceptual System,* the *Motor System,* and the *Cognitive System.* Each of the subsystems is characterized as a processor and a set of memories that comprise a memory hierarchy. The most important characteristics of the memories are:

μ = storage capacity measure in items,
δ = decay time of an item,
κ = primary encoding technique, {physical, symbolic, semantic}.

Processors are characterized by

τ = cycle time

which includes memory access time.

The perceptual processor has the entire human sensory system as input, but relies almost exclusively on the eyes.

The motor processor is responsible for arm-hand-finger and eye-head movements. These movements are made in response to the symbolic information encoded in working memory and are dictated by the cognitive system.

In the simplest of tasks the cognitive system passes control from the perceptual system to the motor system with a minimum of involvement. However, the complex task of design requires the cognitive processor to intercede in support of learning, retrieval of information, and problem solving.

The most important principle from the Model Human User's principles of operation is the Problem Space Principle.

"The rational activity in which people engage to solve a problem can be described in terms of

1. a set of states of knowledge,
2. operators for changing one state into another,
3. constraints on applying operators, and
4. control knowledge for deciding which operators to apply next."

The cognitive model [Mill79, Gaun82] offers additional insight into human knowledge, communication, and thinking.

2.2.4 Computer Aided Communication - *FORUM*

Researchers at the Institute of the Future were interested in improving the process of structured expert interaction [Vall75, Vall77, Joha78], but expanded their domain to include the more general topic of computer-based group communication with *experts* representing but one segment of the user community.

Before developing the FORUM system as a test bed, a series of conferencing case histories were studied and five styles of computer conferencing were identified: (1) the notepad, (2) the seminar, (3) the assembly, (4) the encounter, and (5) the questionnaire.

Having identified the fundamental styles of computer conferencing, the group focused on elements of group communication, such as individual and group characteristics, skills, and attitudes, characteristics of the medium, and communication tasks to be performed.

FORUM tracks the content of discussions, allowing the roles of different participants to be studied. At least three distinct roles are identified, the *provoker,* the *synthesizer,* the conference *leader,* and the conference *facilitator.* The *provoker* seems to push the discussion forward into new areas of thought, while the *synthesizer* ties the loose strands together. The conference facilitator is characterized by participation in a large number of private communications with individuals. Many of these messages are attempts to acclimate the correspondent with computer conferencing and to conflict resolution. The conference leader sets the tone of the conference and controls it with a style ranging from democratic to authoritarian. The leader is

characterized by a high degree of public messages.

After monitoring several real-world conferences conducted using FORUM, the authors express optimism about some aspects of computer conferencing and skepticism about others. The topics of the conferences include international crisis management, mineral wealth forecasting, astronomical data resource sharing.

Many of the performance characteristics presented are based on keyboard input and character display output devices. Positive conclusions include: faster rate of information exchange compared to face-to-face conferences, user satisfaction not determined by previous typing or computer skills, and user participation on a more equal footing compared to face to face conferences. Another significant conclusion is that the conference style is strongly determined by the conference leader as in face to face conferences; the long term impact on the using organization can not yet be determined.

2.2.5 Data Base Management

Landis [Land86, Land83], in a companion dissertation, discusses at length the requirements of a data base for an automated design system. She identifies relational data base technology as the most supportive of those requirements. The requirements that distinguish a design data base from the conventional data base found in business applications include the unpredictable length of fields within records, the granularity of data base access, and the data base transaction unit.

Other researchers are drawing the same conclusions and have proposed changes to existing relational data base systems to allow them to support CAD systems.

2.2.5.1 SYSTEM R - IBM

System R is a relational data base management system developed by IBM Research Labs in San Jose, California. Continued research includes the modification of System R to improve its ability to handle design data [Astr76]. The extensions focus on four major areas of concern: the handling of complex design objects, the support of conversational transactions, the data base – data structure interface, and the management of data items of unlimited or unpredictable length.

The ability to handle complex objects often found in the design environment is enhanced by allowing a designer to declare and specify structural relationships among semantically related data. Such complex objects might be the representation of a VLSI chip with its functions, components, pins, etc. In order to handle conversational design transactions, the capability to provide protection and sharing of a complex object was added. Other modifications provide an object oriented interface which allows a data structure to be synthesized from one or more complex objects. System R's facilities for managing design data are being modified to enhance performance on updates and to support items of variable length.

These extensions to System R have allowed work to proceed in the management of data for a large internal design system. This modified System R can be used to support design at all levels.

2.2.5.2 INGRES - UC Berkeley

Researchers at the University of California, Berkeley have been experimenting with implementation of a CAD application with INGRES, a relational data base management system [Held75a, McDo74, Held75b, Ston76]. Their work has identified four features which need to be added to INGRES in order to more fully support CAD

applications. These features include facilities for the support of ragged relations, support of transitive closure, access by spatial location, and unique identifiers.

Ragged relations are needed to allow for repeating fields within a relation. Often in the design process the need for multiple values associated with an attribute arises and is best represented in relations as repeating fields. The approach to support ragged relations is to allow for ordered relations and then allow the relations to be nested. Transitive closure support is necessary in order to exploit the hierarchical nature of design, such as that which occurs in the expansion of subcells within a cell. Access by spatial location further integrates the geometric aspects of design along with its textual aspects. In order to truly integrate a data base, unique identifiers are deemed necessary for all data base objects.

These enhancements to INGRES, the researchers claim, allow for a data base management system which would be easy to use and which would perform at acceptable levels. INGRES could be used to provide data management support at both high and low levels of design.

2.2.6 Syntactic and Semantic Specification

There are several well known methods of language definition techniques that are sufficiently powerful to describe the interface syntax of interactive systems. A desirable property of a language definition technique is the ability to automatically generate a recognizer, or parser, from the language definition. Much formal work has been done by researchers in the formal language area. Space and time efficient algorithms and language representation are well known and understood. Lewis, Rosenkrantz, and Stearns [Lewi76] provide a thorough discussion of current techniques in this area.

Another necessary property is the ability to associate semantic operations with recognized sentential forms or sentence fragments. A successful technique would thus allow:

1. syntactic definition

2. semantic definition

3. generation of space and time efficient parser that implements the syntax and semantics of the specified language

We will later impose further requirements on the technique we choose.

2.2.6.1 Natural Language

Perhaps all systems are first described in natural language. Natural languages are far more powerful and flexible than any artificial languages that have been introduced. However, this flexibility prevents us from parsing the language by space and time efficient techniques.

2.2.6.2 BNF and Knuthian Semantics

Knuth provided some early discussion of left to right parsing techniques [Knut65] and subsequently adding semantic actions to the parsing techniques [Knut68]. The ability to parse programs written in these languages stems from some severe restrictions imposed on the language's grammar. These restrictions were refined in later work on LR grammars.

2.2.6.3 LR(k) and S-attributes

The LR family has several members and a great deal of formalism and techniques that do meet requirements so far identified. Grammars of the LR(0) class [Aho72, Aho73] and SLR(1) [Reme71] can be recognized by a technique known as SHIFT-REDUCE processing. SHIFT-REDUCE processors strike an effective compromise between table storage requirements and speed of processing. They also have a property that makes them able to provide meaningful error messages and to recover from errors. Both LR(0) and SLR(1) grammars are properly covered by LR(1) grammars. A thorough discussion of the space and time performance characteristics of SHIFT-REDUCE parsing is found in [Aho72, Aho73].

Fenchel [Fenc78] uses SLR grammars to parse interactive user input and to provide automatic syntactic help.

LR(1) is more general than LR(0) and SLR(1). It represents a broader class of languages, but requires considerably more processing power. LR(k), where k is greater than one, is even more general, but in a practical sense provides no additional power over LR(1).

LALR(1), as shown in YACC [John75, John78], provides almost all of the generality of LR(1) with far less processing requirements.

The LR family is discussed thoroughly in [Weth81]. When semantics are added to LR grammars they are termed S-attributed grammars.

2.2.6.4 LL(k) and L-attributes

The LL family is less expressive than LR and approximately as expressive as SLR [Beat82], that is, LR can generate languages that LL is unable to express. LL grammars can be parsed efficiently in terms of space and time. They can provide excellent error messages and error recovery. They possess the property of incrementality; a new grammar rule can be added without reanalyzing the entire grammar. They also possess an interesting property that we may exploit, *prescience*. At any point in the parse, the parser has available to it all inherited and synthesized attributes with which to guide the parse. In general, this property allows LL to perform translations impossible in LR. In particular, the parser knows precisely which token to expect next, and we may wish to exploit that knowledge when we have multiple input devices and must enable the correct device.

The parsing technique is simple and efficient. When semantics are added to LL grammars they are termed L-attributed grammars.

2.2.6.5 Integral Help

Interactive tool users bring with them varying degrees of sophistication in general computer use and varying degrees of familiarity with a specific tool. A tool system's ability to cope with error prone users by detecting and reporting errors can contribute to its usability. Self documenting systems, those that provide on-line syntactic and semantic help, are far easier to learn and use than those that don't offer such facilities. Fenchel's work is representative of the class of self-describing systems [Fenc82].

Fenchel [Fenc78, Fenc80] offers a structured technique that guarantees consistent and accurate syntactic assistance and provides a framework for integral semantic assistance. The technique requires the specification of the language with which a user interacts with the system as an integral part of system design and implementation. The language is processed to produce a parser as well as on-line assistance information, error messages and hard copy user manuals.

Requirements for such help systems include:

1. assistance describing the structure and meaning of their commands and directions for their overall use.

2. consistent availability of assistance,

3. accurate information and reasonable cost for developing and maintaining assistance information.

CHAPTER 3

The Design Methodology and System Organization

This chapter provides a description of the Constructive Design Method, CDM, to be followed during the conceptualization, design, and construction of CADOCS tools. The method is four-phased. Each phase is supported by a language and a language analyzer. The system of language analyzers is described. Input, output, and processing requirements are given for each language analyzer.

3.1 Introduction

This dissertation defines a method that carries the designer of a new tool from the conceptual definition of that tool through to its actual implementation. A method should be easy to follow and attractive to use without assuming that the tool designer intends to use a system, such as IDEAS, that fully or partially automates the design process. If the designer does intend to use such a system, application of the method should provide the designer an appreciable assist in tool construction by offering guidance in describing the tool, by automatically building major portions of the tool, and by providing design feedback.

Many tool developers use general purpose methods to define CADOCS tools. Structured design is an example of such a general purpose software method. The method described here is specifically for CADOCS systems and is considered to be within the CADOCS system. The method is constructive, that is, its use leads the designer to a better understanding of the modeling capability being proposed.

Frequently, user manuals are developed first and used in customer/client discussions. This focuses the designer's and the user's attention on the syntax of the modeling tool rather that on its semantics. CDM specifies the semantics first and separately. The user-manual-first approach prematurely commits the designer to a single user interface. The interface specified in the user manual may make or break the system. This points out a strong need to decouple a tool's syntax from its semantics, and to provide for separate design and review of each. The constructive design method provides for that partitioning.

Many tools are designed assuming that a specific device or set of devices will provide the means for tool interaction. The method separates physical device considerations from semantic and syntactic considerations. The design process is obviously more complex if a tool designer must consider all of these issues at once, and, in fact, a single designer may not be qualified in all of these specialized areas. It may be advantageous to have the modeling capability definer and members of the user community develop the tool's concept, to have a specialists in human factors and user interfaces focus on that portion, to have graphics programmers develop device handlers, and to have applications programmers develop the body of the semantic routines (operations).

The method is four-phased and will be outlined in the remainder of this chapter. The method and languages to support each phase will be described in detail in a later chapter as will the execution environment for CADOCS systems.

3.2 Imprecise Goals of the Method and Kernel

The following list of high level goals is imposed on the tool development method and underlying support kernel.

1. Support strong partitioning of user interface issues from tool modeling issues.

2. Automatically provide a consistent, friendly user interface directly from the tool's specification.

3. Provide a high degree of device independence.

4. Provide a large body of reusable software that is common to CADOCS tools.

5. Automatically generate as much code as practical.

6. Adhering to the method should produce a better tool more easily than not following it.

7. Method should be constructive. It should lead to a better understanding of the new tool during specification instead of later during coding.

3.3 The Design Method

The method is multilevel and object-oriented. The four phases of CADOCS tool design method are the *conceptual, grammatical, logical device* and *physical device* phases.

The conceptual phase supports a designer in the creation of a new modeling capability from its initial conceptualization to its complete semantic definition. Modeling objects, their inter-relationships, and the operations performed upon them are all identified in this phase.

The grammatical phase supports complete syntactic definition of a tool. Primitive grammatical elements, words (terminals), are identified. Terminals can be ordered and collected into phrases and sentences (non-terminals). The tool designer

will associate each sentence with a semantic operation defined in the previous phase. The run-time environment must support the recognition of sentences and an invocation of the specified operation.

The logical device phase focuses the designer's attention on the logical devices and logical interaction tasks required to support interaction. A handful of logical devices (e.g., selector, valuator, pick, text) have been identified by Foley and VanDam [Fole82]. Software implementations of these logical devices are developed for available physical devices (e.g., specific mice, tablets, keyboards). These logical device drivers are maintained in a special library. This phase is primarily concerned with the mapping of words from the previous phase onto logical devices from the library.

Finally, the physical device phase focuses the designer's attention on the physical device inventory available to support interaction. Logical devices and tasks are assigned to physical devices. Several other aspects of the physical environment may also be dealt with in this phase.

3.3.1 The Conceptual Phase

The first step is the identification of objects that the user of the tool must be familiar with. These are the objects that the user will create and manipulate during use of the CADOCS tool. The relationships that must hold between objects must also be defined. Lastly, the operations upon objects are specified.

Objects emanate from the tool and from the support environment provided for the tool. Those that are derived from the functional specification of the tool must be identified and developed by the tool designer. They are those objects that are specifically part of the tool or set of tools under development. Other objects are present across a broad spectrum of CADOCS tools. A display screen is such a

general purpose object. The reason we distinguish between tool specific objects and environment objects is that we wish to provide help to the tool designer in building reusable environment objects. We can provide higher level support for those objects that recur in CADOCS environments than we can for tool specific objects.

A most important object that recurs across tools is the CADOCS data base. It is included as an environment object because of its frequency of appearance and the complexity of its effective use. We have tentatively selected a relational data base and have integrated it into the kernel as well as into the tool design method. Applying the method via the higher level constructs provided in the specification language relieves the tool designer of the task of constructing a special purpose data base.

Operations come in several varieties. Validation operations are pure boolean functions. These operations determine legitimacy of proposed operations on the collected objects but can have no side effect on those objects. Query operations extract certain information from the underlying collection of objects, and have no side effect. Transformation operations are those that alter the objects that they operate on. A complete list with definitions will be developed in the next chapter.

The specification of objects, relations, and operations are performed only at the user level to facilitate customer/client review. This approach can be likened to that of the use of Abstract Data Types, ADTs. The Ada programming language supports this design principle through Ada package specifications parts.

At this point, the tool implementation team may split into groups. One group can begin further specification and implementation of the operations previously defined. Another group may concentrate on user interface issues.

3.3.2 Grammatical Expression of Dialogue

The tool designer is concerned with the tool's user interface during the grammatical phase having defined the tool's semantics in the previous phase. Preliminary research has identified only a portion of the requirements for this phase of the method. The obvious requirements include the syntax of the tool user's inputs, the tool's semantic response to inputs, and the form of presentation of modeling objects produced during tool use. Since display screen layout and menu presentation are significant aspects of user dialogue, they must also be considered during this phase.

3.3.3 Logical Device Level Considerations

The logical device phase focuses the tool designer's attention on the low-level aspects of the tool's user interface. None of the previous phases have specified the characteristics of any specific device. Each primitive notion identified in the previous phase must come from some physical input device and be displayed on some physical output device. This phase buffers the designer from the specifics of physical devices with the more general interface characteristics of *logical devices*.

An example of a common logical device is the pick device. A user interface that requires its user to pick a modeling object from those displayed on a screen or one of the menu alternatives presented would find a logical pick device a natural and powerful aid. The method allows the tool designer to *bind* words from the previous phase to logical devices offered by this phase.

3.3.4 Physical Device Level

The physical device phase allows the tool designer to specify the physical device inventory of interaction devices that are available to support the tool's user inter-

face. A remaining step is to bind logical devices to physical devices.

3.4 Overview of the System Organization

Each step of the methodology is supported by a specification language, language translator(s), and analyzer(s). The following observations serve as rationale for this approach.

1. Each design decision reached should be documented.

2. Documented decisions require a vehicle for review.

3. Once a phase is specified, some level of confidence must be achieved before attention is shifted to the next phase.

4. A subset of the decisions made in one phase are useful as input to subsequent phases.

5. Many of the decisions made must eventually find their way into implemented software. This migration into code should be supported by code generation or by verification of specification against implemented code.

The specification languages are defined to directly support the methodology outlined in this chapter. The following sections identify the inputs and outputs of the specification language processor for each phase. A complete description will be given for each language (input requirements for the language compilers) in the chapter devoted to that phase of the methodology. All output languages will be similarly defined for each phase. These serve as interface specifications for the various phases.

Separate languages are provided for:

a. Tool Conceptualization

b. Dialogue Description

c. Logical Device Description

d. Physical Device Description

Figure 3.1 shows the data flow aspects of system generation.

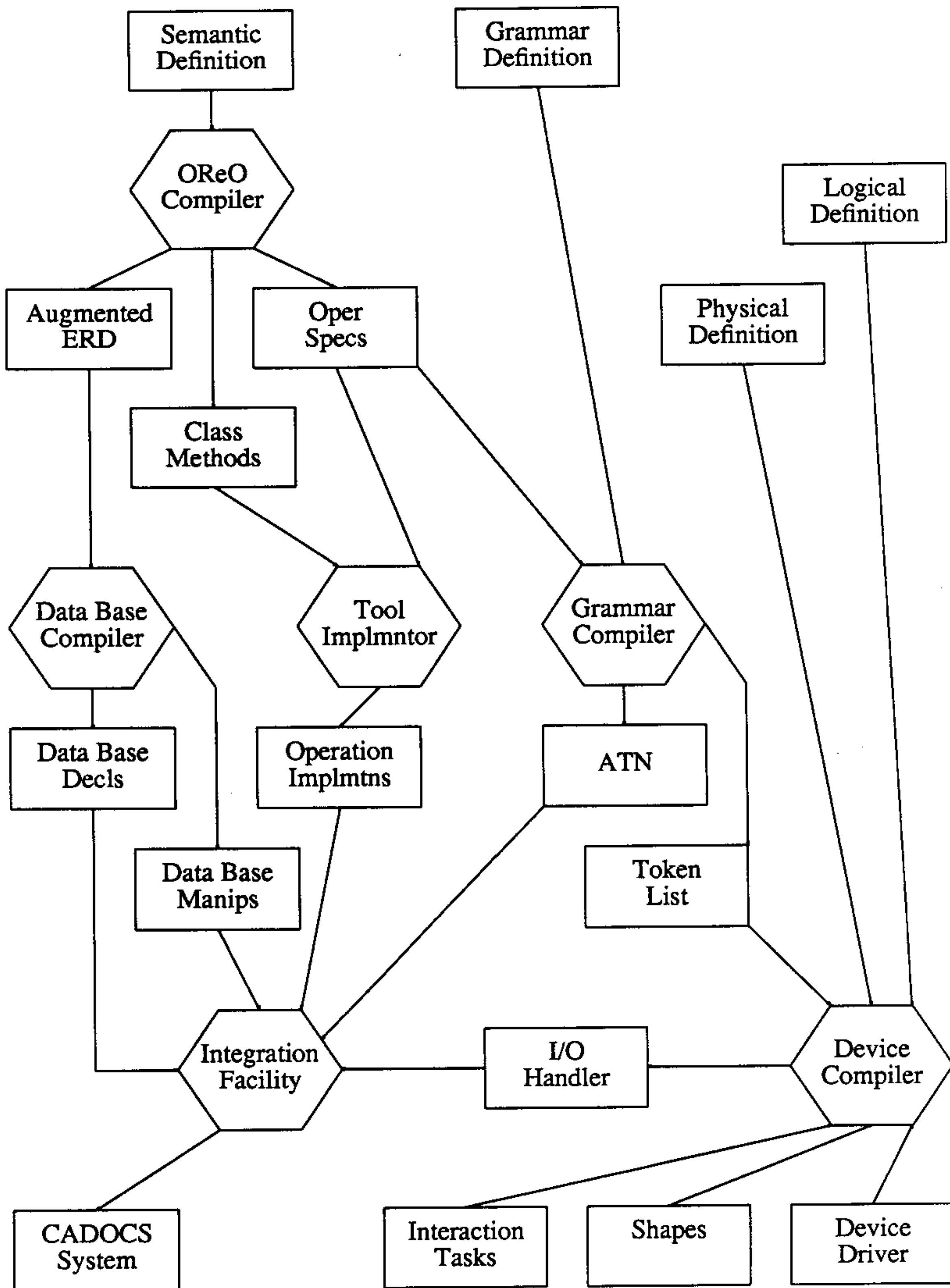


Figure 3.1: System Generation Data Flow

3.4.1 Conceptual Language and Support Software

The description fed into this facility must be sufficiently rich to compute an Entity Relation Diagram, ERD, of the tool. The facility must also compute other data structures or procedures that are not part of ERDs yet are essential to our goal of generating a complete data base management component. The original ERD and these augmentations comprise an Augmented Entity Relation Diagram. Figure 3.2 represents the data flow aspects of this portion of System Generation.

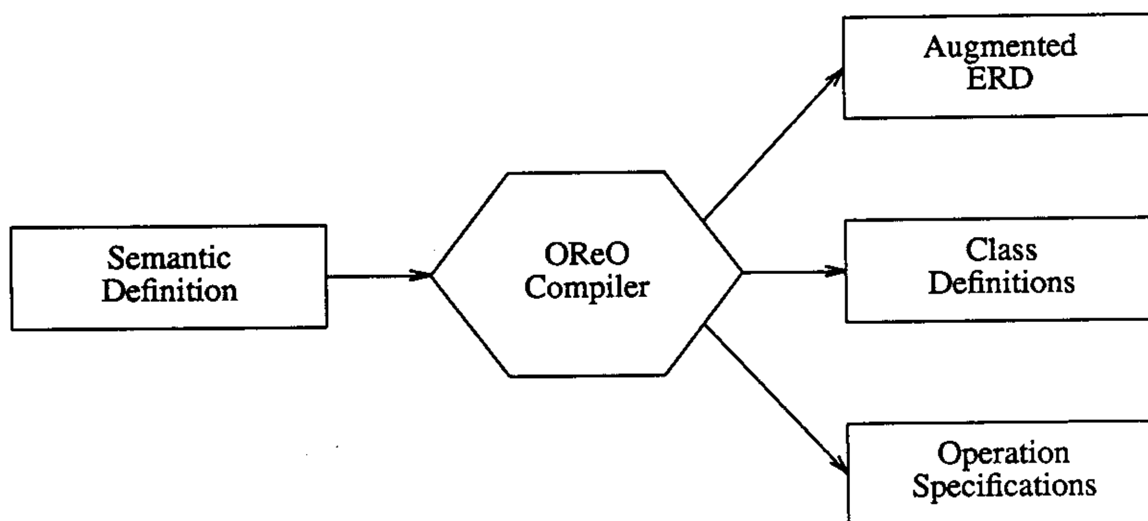


Figure 3.2: OReO Compiler

PROCESSING

1. Semantic operation specifications are essentially copied from input to output unscathed.

2. Generate class definitions and methods sufficient to describe in-memory models that the tool will construct and manipulate at run-time.

3. Generate class definitions and methods sufficient to describe data base models that the tool will construct and manipulate at run-time.

INPUTS

1. objects seen by the modeling tool's end user,

2. relations

   a. internal relations - those relations existing between objects defined within a single modeling tool,

   b. external relations - those relations existing between objects not defined in a single modeling tool,

3. operations

   a. transformation operations

   b. functional (query) operations

   c. pre-transformation validation operations

   d. time-invariant validation operations

OUTPUTS

1. semantic operation specifications,
2. in-memory descriptions and manipulations of modeling objects,
3. database descriptions and manipulations of modeling objects,
4. augmented ERD

3.4.2 Dialogue Language and Support Software

This phase primarily involves the ordering of input tokens. Sentences of the user interface are defined at this point using a BNF-like meta-language. Operations to be performed upon recognition of sentences are also identified. Input validation operations are interspersed within the sentences. Error recovery steps to be taken upon bad input are also specified.

The meta-language is L-attributed. The dialogue specification as well as the operation interface specifications from the previous phase is input to the interaction compiler. This compiler is responsible for assuring adherence to the LL(1) restriction, and for assuring that only previously defined operations are invoked. It is hoped that certain metrics can be computed by this compiler that measure the quality or consistency of the interface. In addition to interface feedback reports, this compiler will produce some representation of the grammar that is suitable for driving the user's interaction with the tool. The form must also be suitable for providing integral help, generation of menus, error recovery, and a host of other capabilities not yet defined. Currently, the Augmented Transition Network [Wood70], ATN, provides the framework for this output. A final output of this compiler is a list of terminal symbols found in the language. These symbols can be regarded as undefined external references that we expect to resolve in a later phase. Figure 3.3 shows the data flow aspects of this portion of System Generation.

PROCESSING

1. Insure LL(1) property.

2. Insure that only legal operations are associated with sentences.

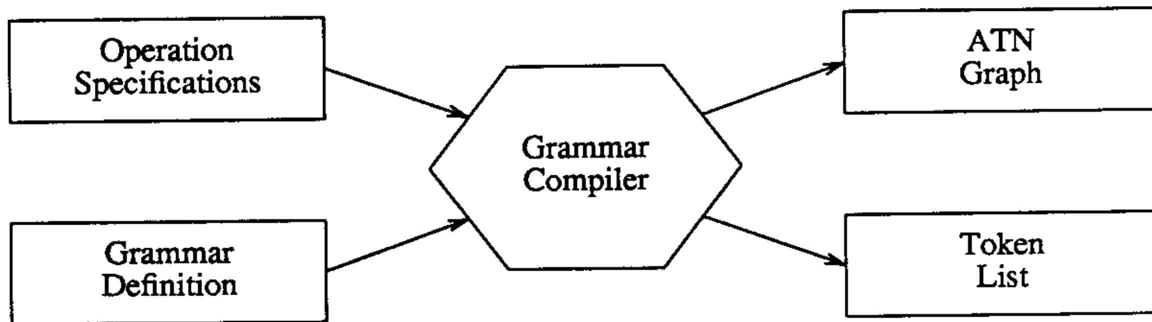


Figure 3.3: Grammar Compiler

3. Insure that operations are called with appropriate number and type of arguments.

4. Construct graph representation of grammar.

INPUTS

1. grammar specification

2. operation interface specifications

OUTPUTS

1. external form of the tool's grammar in a form that can be subsequently read and traversed.

2. list of grammar's terminal symbols

3.4.3 Device Languages and Support Software

The lexical and physical device phases are characterized by a device compiler. The input specifications of this compiler define the lexical device and physical device languages. Another input to this compiler is a list of terminal symbols produced by the interaction compiler. With these inputs and access to the logical device library, the compiler is responsible for producing an input/output handler for the executing tool. Figure 3.4 shows the data flow aspects of this portion of System Generation.

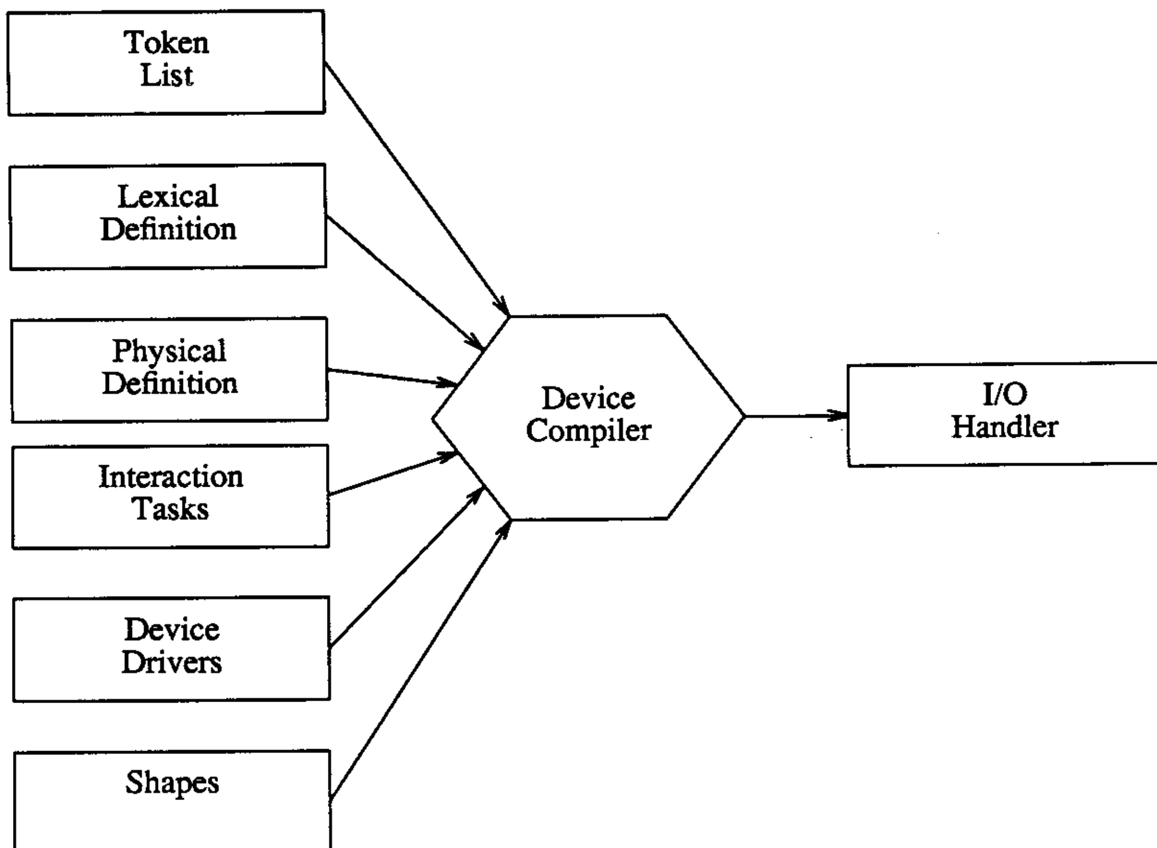


Figure 3.4: Device Compiler

PROCESSING

1. Translate inputs into requests to the system librarian for DDL support.

2. Translate inputs into requests to the system librarian for DISL support.

3. Translate inputs into requests to the system librarian for LDL support.

INPUTS

1. lexical definition of the tool,

   a. binding of grammatical input tokens to interaction tasks,

   b. binding of grammatical output tokens to shapes,

2. physical device requirements of the tool,

   a. physical device inventory,

   b. binding of interaction tasks to physical devices,

   c. binding of shapes to physical display devices,

3. interaction tasks, software implementations of a logical interaction for a specific physical device,

4. device drivers,

5. graphical shape definitions,

OUTPUTS

1. software implementation of the required I/O handler.

3.4.4 Entity Relation Diagram

The ER diagram can be used as input to a separate tool responsible for augmenting the run-time data base management system. It is anticipated that the information typically specified separately in a data description language, DDL, and in a data manipulation language, DML, can be automatically extracted by the OReO compiler. This descriptive activity is typically performed by a data base administrator, DBA, as a separate design activity and requires specialized expertise.

The DB compiler is the processor within OReO which is responsible for the coupling of the tool and the data base management facilities. Figure 3.5 shows the data flow aspects of this portion of this of System Generation.

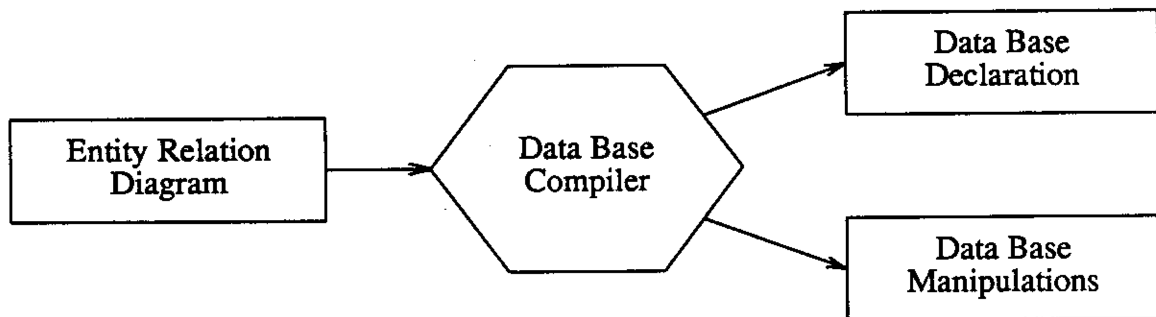


Figure 3.5: Data Base Compiler

The tool's physical, as well as the logical, data base must be declared including relationships with external entities.

PROCESSING

1. Traverse the augmented ERD to identify types of entities, relationships and properties.

2. Map entities, relationships and properties into the RM/T data model.

3. Map the RM/T model into relations supported by underlying relational DBMS.

4. Map the constraints, options list, defined unit and defined methods into catalogs and procedures supported by the DB Kernel.

INPUTS

1. The ERD (Entity-Relationship Diagram) consisting of defined entities:

    a. properties of entities

    b. relationships between entities

    c. roles entities assume

    d. entity origin

2. The ERD augments consisting of:

    a. type constraints on entities,

    b. value constraints on entities,

    c. structural constaints on entities,

    d. procedural constraints on entities,

e. unit definition

f. options list

OUTPUTS

1. DB declaration of catalogs which define the RM/T model,

2. DB declaration of catalogs which define the external links, and

3. DB methods which augment the T methods provided by the DB Kernel.

3.4.5 Auxiliary Support

The integration facility is a collection of software that takes the several results from the phase-specific tools and brings them together into an executable program equivalent to the tool specification. It is in a position to perform other ancillary duties.

The tool developer is responsible for fleshing out the operation specifications identified in the conceptual phase. A consistency checker may be required to guarantee agreement between the semantic operation's specification and implementation. Other analyzers may be identified that either support verification or interface analysis.

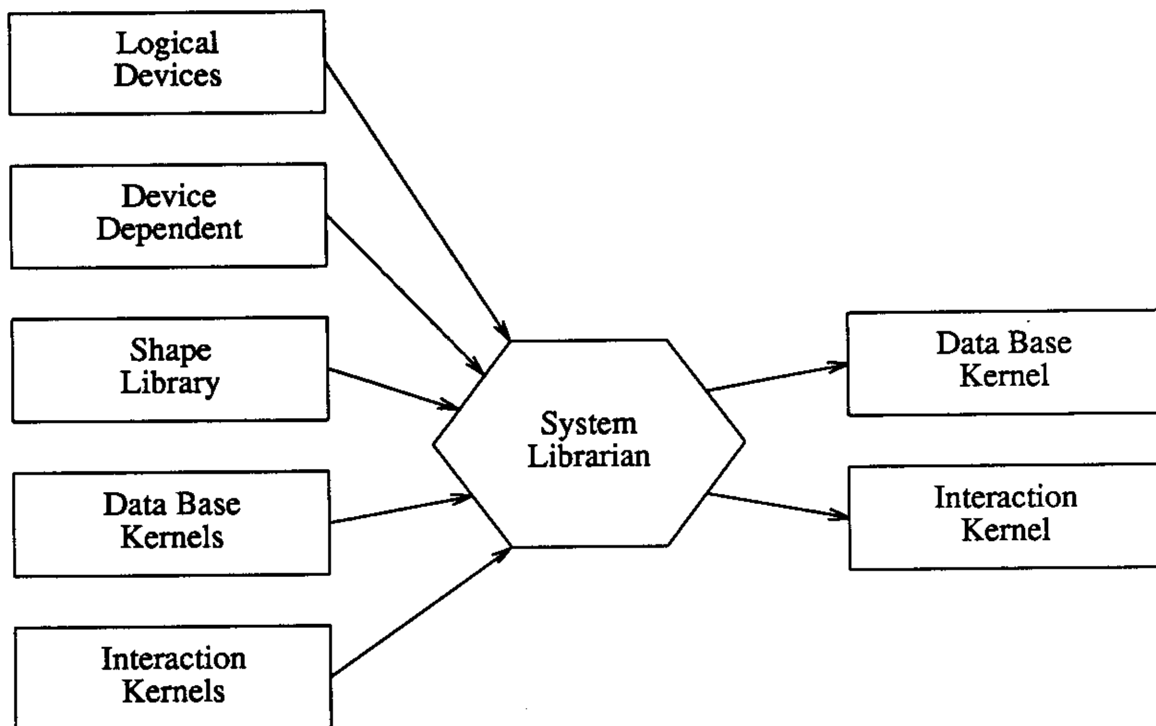


Figure 3.6: Integration Facility

PROCESSING

1. Verify the consistency of operation specification and implementation.

2. Construct an executable product from the constituent fragments and determination of completeness.

INPUTS

1. data base management inputs

   a. static (run-time invariant) declaration of the tool's data base,

   b. methods that can be applied to the data base at run-time,

   c. kernel (tool independent) data base software.

2. syntactic and semantic inputs

   a. interface specifications of user available operations,

   b. implementations of those specifications,

   c. parse table or its logical equivalent.

3. miscellaneous inputs

   a. an implementation of the tool invariant interaction kernel,

   b. a tool dependent and device dependent I/O handler.

OUTPUT

1. executable CADOCS system tailored to some hardware environment.

3.4.6 System Librarian

The System Librarian will also be part of the RUN-TIME environment component. It will also be accessible as a stand-alone processor. The focus here is on the role it plays during Sysgen. It is essentially a data base management system that manages libraries of data structures and procedures that are used in the construction of CADOCS systems.

The librarian will accept requests to search the various libraries under its control, requests to install new library entries, requests to query properties of the library entries. The librarian will also perform processing normally associated with version control.

Inputs to the System Librarian are in the form of requests submitted by other components of the SYSGEN environment. The run-time environment may also submit requests.

The managed libraries are also inputs to the librarian. The libraries are currently (very subject to change)

1. the Logical Device Library (contains sets of logical input routines, in each set is an implementation of each interaction task, there is one set per physical input device),

2. the Device Independent Shape Library (contains routines to draw shapes or icons, these routines are implemented as calls upon routines that draw generalized graphics routines and are thus device independent),

3. the Device Dependent Library (contains sets of output routines, one routine per generalized graphics primitive comprises the set, one set per physical

display device),

4. the Data Base Kernel Library, and

5. the Interaction Kernel Library.

The System Librarian will normally output source or object level descriptions of data structures and procedures in response to requests. It will respond with directory-like information for appropriate queries.

3.5 Summary

A separate chapter of the dissertation is devoted to the detailed exposition of each phase of the method. The associated specification languages, translators, and analyzers will be described in the appropriate chapter. The translators each require certain primitive run-time support if we are to provide auto code generation. This run-time support, the target machine, is developed and defined in the chapter on the system kernel.

A later chapter will give one or more design examples that demonstrate all phases of the method. Examples will show definition of the first tool in the system and the later inclusion and integration of another cooperating tool.

CHAPTER 4

Conceptual Definition Phase

Chapter 3 introduced the four phases of tool definition, concept, syntax, logical device, and physical device level definition. This chapter is a complete discussion of the concept definition phase. The design methodology relevant to this phase is made more concrete by applying it to the description of a simple tool, the Influence Diagram Editor.

After describing the fundamental characteristics of the influence diagram, each step within the concept definition phase is applied and specification fragments given. Later, the entire specification language is given along with the complete concept specification of the Influence Diagram Editor.

During the conceptual definition phase the tool designer initially focuses attention on object identification, then on the relationships that exist between objects, and finally on the operations performable on the objects.

4.1 Influence Diagrams

Influence diagrams have been shown to be useful in a variety of guises. They appear to be used first to model continuous systems such as the growth patterns in a large urban center [Forr61, Forr69]. Later they reappear in the form of cognitive maps as exemplified by [Bonh76].

Influence diagrams describe the ways in which concepts influence each other. The standard diagram includes text strings, sometimes individually enclosed in a circle or an ellipse, that represent *concepts*. Typical concepts from Forrester's Urban Dynamics, for example, are population, industrial base, number of homes, and perceived quality of living. Perception of a high quality of living in an urban area tends to cause an influx of new arrivals from rural and other urban areas. The perceived quality of living is in direct proportion to the rate of population growth. More than just positive or negative correlation is captured in an influence diagram. Cause and effect are also of interest. A change in the perceived quality of living causes a like change in the rate of population growth. The cause and effect and the correlation are represented by the *influences*. Influences are directed, labeled arcs between concepts. The arrow on an influence points to the affected concept. The labels "+" and "-" are used to signify positive and negative correlation influences respectively. Forrester attempts more precision by attaching some mathematical function to the influence to represent the weight of the influence.

Figure 4.1 is a small influence diagram taken from the urban dynamics domain.

Coyle has applied influence diagrams to military analysis. He adds time delays and non-correlated influences, but attempts no mathematical expression of the

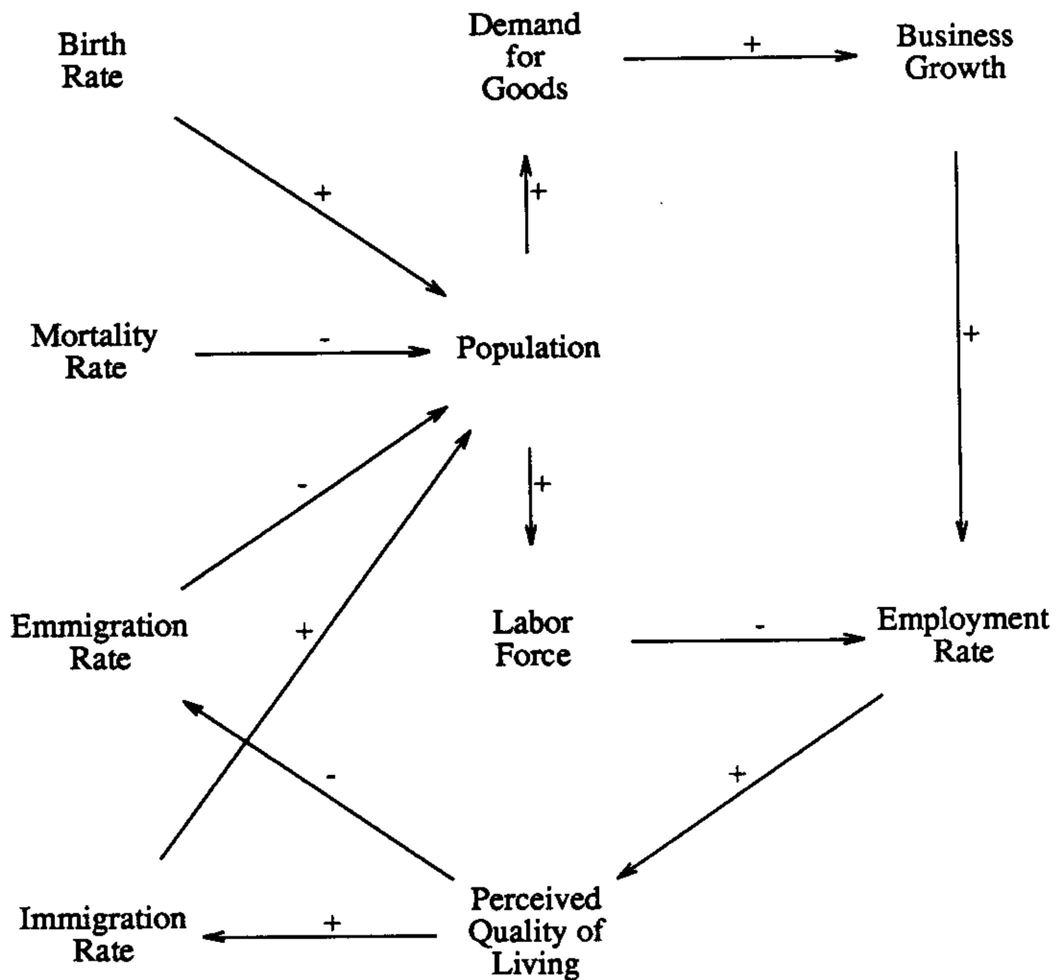


Figure 4.1: Forrester's Influence Diagram

influence. Figure 4.2 is a simple influence diagram taken from Coyle [Coyl81].

Influence diagrams have been called cognitive maps when used to represent the belief systems of high-level policy makers. The influence diagram begins at the top with policy alternatives (concepts) and ends at the bottom with some abstract utility concept that represents the well-being of the policy domain. Immediately above the utility concepts are abstract components that comprise the utility concept. In between the policy choices and the utility concept are the cognitive concepts of the

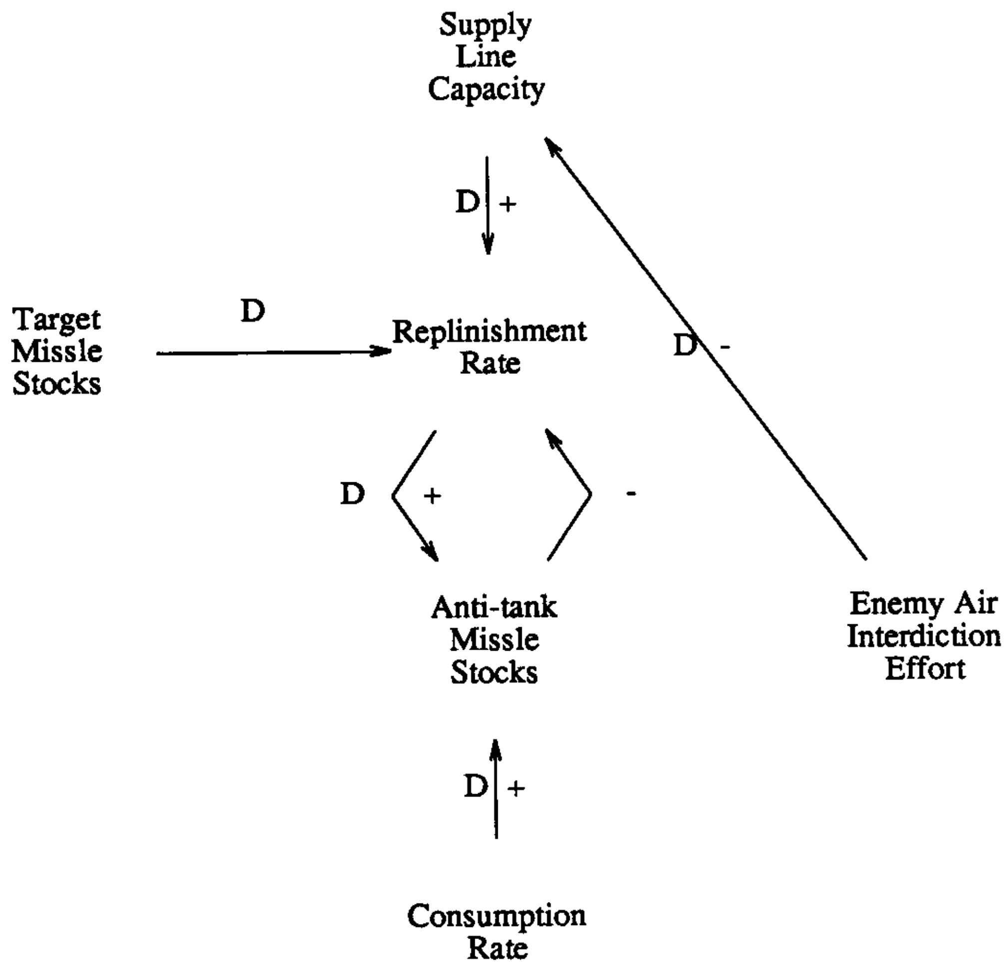


Figure 4.2: Coyle's Influence Diagram

policy maker or of the society. Each concept influences, or operates in relation to, other concepts in the diagram. No attempt is made to weight the influences in a cognitive map.

The cognitive map has been adapted for use in strategic planning by Worley [Worl85]. The plus and minus symbols have been replaced by shaded circles. Solid circles are used to indicate a positive correlation, half filled circles indicate a negative correlation, and non-filled circles indicate no correlation at all. Research shows that

strategists confuse plus with increase and minus with decrease instead of with positive and negative correlation. Cognitive maps represent a lack of correlation between concepts by showing no influence. Strategists often explicitly express that no correlation exists between certain concepts. Hence, the addition of non-filled circles to make non-correlations explicit.

Figure 4.3 shows such a cognitive map. The top center concept of the map represents a Soviet policy alternative, the invasion of Europe. A war in South West Asia (SWA) is believed to increase the likelyhood of a Soviet invasion of Europe. The Soviet decision maker believes that invasion of Europe will not lead to intercontinental (IC), nuclear exchanges. The utility concept that represents the decision maker's objective is an easy gain of territory in Europe. That utility concept is refined into three other cognitive concepts, a short war, a conventional (non-nuclear) war, and a narrow war involving only West Germany, Denmark, and possibly the Benelux countries.

4.2 Object Identification Step

First we identify objects that the user must be aware of. These objects are viewed and manipulated through tool use. The end user of the Influence Diagram Editor (IDE) can view the tool as an object with which to interact. The end user will also recognize some other objects within the tool. First on our list is to describe those objects that the end user sees and eventually manipulates via the user interface.

The object identification step section of the specification yields a collection of single object identification paragraphs separated by blank lines. The elements of an object paragraph are the topic sentence, a population sentence, and optionally some descriptive text sentences. The topic sentence provides the object's name and possi-

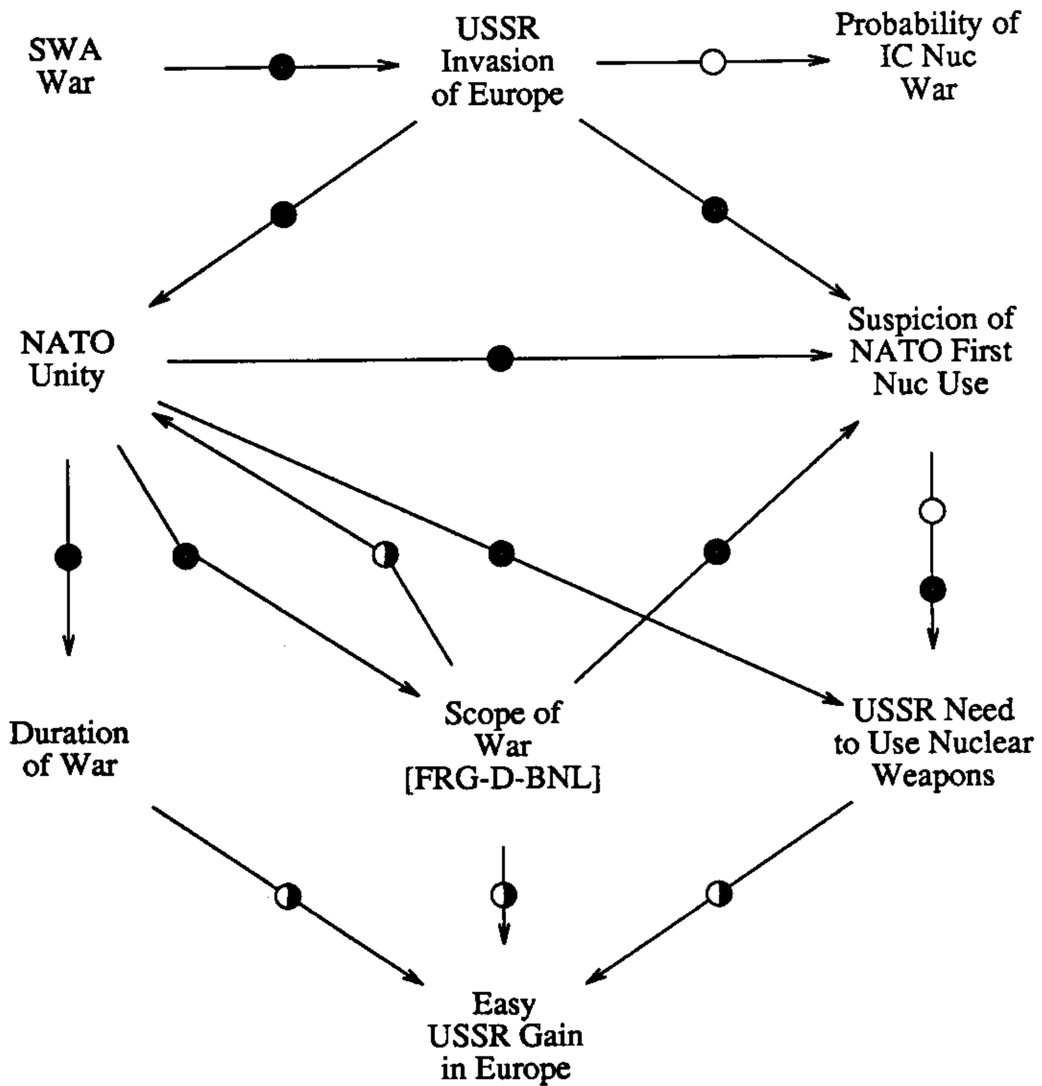


Figure 4.3: Worley's Influence Diagram

bly a synonym. The population sentence indicates the number of this type of object that exists when the tool is initiated and the number of objects that may occur throughout tool use. The object identification paragraph may also include some arbitrary text that is useful to the end user and to the tool implementor.

The objects that comprise an influence diagram are the influence diagram itself and concepts. Typically, an end user needs to know things like: Is there some fixed number of influence diagrams throughout IDE use? How many concepts are there after a complete IDE use session? How many concepts are there upon initiating an IDE dialogue? In addition to providing answers to those questions, the tool designer may give synonyms for the objects' names. Synonyms may be introduced for a variety of reasons, but primarily to provide the plural name. Having both a singular and plural object name allows the tool specification to read more naturally, but is otherwise unnecessary. The tool designer may also provide descriptive text that is useful to the tool implementor and that is retained as semantic help for the end user.

The object definition for the influence diagram and the concepts are provided below as example.

The object *Influence Diagram.*

> The influenceDiagram is an object. Initially, there are none; later, there is 1. Initially, the end user will not be provided with an empty influence diagram. It is the user's responsibility to create it. Only one influence diagram at a time will be allowed during Influence Diagram Editor use.

The object *Concept.*

> A concept (concepts) is an object. Initially, there are none; later, there may be many. Concepts may represent real-world events, human beliefs, or rather vague, non-quantifiable goals. Concepts influence each other.

4.3 Relation Identification Step

Identifying objects is a useful, easy first step, but hardly sufficient to describe a tool to an end user or to a tool implementor. The objects so far defined enter into relationships with each other. Our next step is to define these relationships. Chen [Chen76] and others have shown that a description of these relationships provide a highly lucid view of any enterprise. The Entity Relation Diagram, ERD, has been applied widely in the field of database management. We enter this step with goals of refining our understanding of the tool under construction and of providing the end user, tool implementor, and database management system implementor with a rich semantic understanding of the tool.

4.3.1 Relation Types

Objects enter into relationships with other objects. There are three types of relationships possible. The differences between types are important but the means of discription for each type is uniform. The type of relation is inferred from the description and relies on recognizing the type of nouns used in the description.

```
<aNoun> : <anObject>
    | <anAttribute>

<anObject> : <aSimpleObject>
    | <anAggregatingObject>

<anAttribute> : <aPrimitiveType>
```

The first type of relation is that which objects enter into with primitive types, called attributes. An example of this type of relation is the one that exists between a concept and its name.

A concept has a name.

Concept is a simple object but name is an attribute. This type of relation is specified as:

<propertyRelationSentence>:
<article> <anObject> <possessiveVerb> <article> <anAttribute>

The next section will discuss attributes more fully. Primitives are system defined types like string and integer.

A second type of relation exists between objects and what are called aggregating objects. The influence diagram is an aggregating object and it enters into an aggregating relationship with concepts. The influence diagram is an example of an aggregating object.

The influence diagram has concepts.

This type of relation is specified as:

<aggregateRelationSentence>:
<article> <anObject> <possessiveVerb> <article> <anObject>

A third and most common kind of relation is that which objects enter into with other objects. An example of this type of relation is the one that exists between concepts.

Concepts influence concepts.

This type of relation has a name, in this case, influence is the relation's name. The syntax of this relation type's specification is more complicated and the reader should refer to section 4.10 for a full description.

4.3.2 Relation Specification

Like object identification, we use a paragraph to describe each relation. Each paragraph has a topic sentence and optionally a body of sentences. Each sentence is roughly of the form

<sentence> : <subject> <verb phrase> <object>.

The <subject> of the topic sentence is the relation being described in the paragraph. The <subject> is the implicit antecedant of the word "it" in the body of the paragraph.

The most simple relation paragraph is the single topic sentence describing the relation between an object and its single attribute or the relation between an aggregating object and an object.

A concept has a name.

The influence diagram has a name.

A more complicated paragraph begins with a topic sentence that introduces a relation and any attributes that it has, and then specifies the objects entering into the relation.

An influence has a proportionality. A concept influences many (m) concepts, and a concept is influenced by many (n) concepts.

Relationships, particularly when discussed in the context of database management, are one-to-one, one-to-many, or many-to-many. The "one" can often be inferred from a sentence by the article preceding the <subject> or <object>. The indefinite articles "a" and "an" and the definite article "the" imply a one-to-n relation. The word "many" preceding an <object> or <subject> implies the obvious.

Since there may be many relationships in a single tool, and many of them may be one-to-many, it will be difficult to keep them straight and a symbolic name for the individual ''many''s may help. For the ERD that is shown later and the database management system that follows, it is often convenient to have a symbol that represents the cardinality of the relationship. One-to-one, one-to-many, and many-to-many relations are depicted on an ERD as 1:1, 1:m, and m:n respectively. The relation specification language allows for a variety of unnecessary words in the verb phrase, ''has'', ''may have'', and ''must have''. They are all parsed as equivalent and are supported only to make the final specification read more naturally.

> The influenceDiagram has a name and has many (m) concepts.
>
> A concept has a name.
>
> An influence has a proportionality. A causal concept influences many (j) affect concepts, and an affect concept is-influenced-by many (k) causal concepts.

The first paragraph defines two relations. The <subject> of the topic sentence is the influence diagram. It enters into a one-to-one relation with the <object> of the first clause, an attribute of type name. The <subject> of the second clause is inferred to be the <subject> of the topic sentence. The influence diagram enters into a one-to-many (1:m) relation with the <object> of the second clause, the object concept. The second paragraph defines a one-to-one relation between the object concept and the attribute name. The third paragraph defines a relation, influence, that has a proportionality attribute. Notice that the single relation, influence, is bi-directional. A concept enters into an influence relation with another concept, but either concept may play the *active role* of influencing the other or the *passive role* of being influenced by the other. It may be apparent at this point that relations are a type of object. They can

have attributes, are visible in the user interface, and can be manipulated (named, deleted, moved, etc.). This discussion continues in the operation identification section.

4.3.3 Relation Specification Conclusions

The types of relations that can be described with the specification language are quite complex. The concerned reader should study the complete language definition in section 4.10. The use of the 1:1, 1:n, m:n notation is made clear in section 4.6.3. The results of processing the relation specification include an augmented ERD for the database management system implementor (see section 4.6.3) and generated software for use by the system and tool implementor (see section 4.6.1).

4.4 Attribute Identification Step

As discussed above, some nouns are too primitive to be considered objects. Those objects are called attributes. The tool designer may have difficulty in deciding where to draw the line between simple objects and attributes, just as it is a difficult task in object-oriented design or programming to decide when to abandon the object-oriented paradigm. This is a choice left to the tool designer and one that may undergo refinement as the tool specification evolves.

To the conceptual definition compiler, the fact that an <object> is an attribute rather than an object can be inferred by observing that all objects have been identified in the object identification step and therefore anything appearing as an <object> in the relation identification step that was not previously identified as an object must be an attribute. An <object> found in the relation identification step but not introduced as an object in the object identification step could simply be the result of the tool designer's spelling error. Rather than having the compiler make such inferences we

chose to require the tool designer to explicitly identify objects and attributes.

The tool-building system provides possibly machine dependent definitions of base types such as string and integer. The tool designer may define new base types aggregated from previously defined base types. Enumerated types are also supported.

The attributes found in the Influence Diagram Editor specification are shown following.

```
A proportionality is one of [not,direct,inverse].
A name is a string.
```

4.5 Operation Identification Step

There are many types of operations that an end user may be conceptually aware of and that the tool definer must support. The categories of operations are transformation, query, pre-transformation validation, and time-invariant validation.

4.5.1 Transformation Operations

Transformation operations are perhaps the most obvious to the end user. Their chief characteristic is that they operate by side effect, that is, they cause state change. An example from a common text editor is the operation to delete a line.

```
deleteLine(lineNumber:in integer; f: in out file)
```

After invoking this operation, the user expects the file to be different. By providing the operation with the line number to delete and the file from which it is to be deleted, the operation will provide an altered file.

The transformation operation **to** specifications needed for the editor are defined similarly.

```
to MakeInfluenceDiagram() returns influenceDiagram
  --- A new, empty influence diagram will be created.
  --- Only one influence diagram may exist at any time.
  --- Returns Nil if it fails for any reason.
  ---   or a new influenceDiagram if it succeeds.
to MakeConcept() returns concept
  --- A new, unnamed, disconnected concept will be created.
  --- Returns Nil if it fails for any reason.
  ---   or a new concept if it succeeds.
to MakeInfluence() returns influence
  --- A new, unnamed, disconnected influence will be created.
  --- Returns Nil if it fails for any reason.
  ---   or a new influence if it succeeds.
to InfluenceDiagramNameSet(ID: in out influenceDiagram; N: in name)
  --- Returns Nil if it fails for any reason.
  --- If it succeeds, the name of I will be set to N and
  ---   the updated ID will be returned.
to ConceptNameSet(C: in out concept; N: in name)
  --- Returns Nil if it fails for any reason.
  --- If it succeeds, the name of C will be set to N and
  ---   the updated C will be returned.
to AddConceptToInfluenceDiagram
  (ID: in out influenceDiagram; C: in concept)
  --- Returns Nil if it fails for any reason.
  --- If it succeeds, C will be added to the collection of
  --- concepts contained in ID, the updated ID will be returned.
to AddInfluenceToConcepts
  (causeC,affectC: in out concept; I: in out influence)
  --- Returns Nil if it fails for any reason.
  --- Both concepts must exist, not already influence each other,
  --- and I must exist.
  --- If it succeeds, I will be added to the concepts and returned.
to ProportionInfuence(I: in out influence; P: in proportion)
  --- Returns Nil if it fails for any reason.
  --- I must exist. P must be a legitimate value.
  --- If it succeeds, P will be added to I and I will be returned.
to RemoveConceptFromInfluenceDiagram
  (ID: in out influenceDiagram; C: in concept)
  --- Returns Nil if it fails for any reason.
  --- ID must exist. C must exist.
  --- If it succeeds, C will be removed from the collection of
  --- concepts contained in ID, the updated ID will be returned.
to RemoveInfluenceFromConcept
  (affectC,causeC: in out concept; I: in out influence)
  --- Returns Nil if it fails for any reason.
  --- The concepts must exist. I must exist and connect the concepts
```

```
--- as indicated. If it succeeds, I will be disconnected from
--- the concepts, and the updated influence will be returned.
```

Lest the reader be left with the feeling that typing these operation names is an onerous task to put upon the end user, remember that we are not describing the syntax of the interface yet, just the concepts, the semantics. The syntax to remove an influence from a concept might be

```
rm -C c42 -I i13
```

or a pick of the influence followed by a menu pick of the remove operation. Regardless, the semantics are the same, and in fact the command interpreter could invoke the same semantic routines when it recognizes either syntax.

In the relation identification section, it was observed that relations are a special type of object. The astute reader may have noticed that the transformation operations included operations on objects, e.g., *concept* as well as operations on relations, e.g., *influence*.

4.5.2 Query Operations

Query operations differ from transformation operations mainly in that they have no side effect. They do not cause state change. They are pure function. Often they are invoked by the user to extract information, e.g., "On what line number is the cursor?" or "What is the name of the pointed to concept?"

An appropriate set of query operations **qo** for the editor are given below.

```
qo conceptName(C: in concept) returns name
   --- Returns Nil if it fails for any reason,
   ---  or the name of C if it succeeds.
qo idName(ID: in InfluenceDiagram) returns name
   --- Returns Nil if it fails for any reason,
   ---  or the name of ID if it succeeds.
```

4.5.3 Pre-transformation Validation Operations

Early in the research, it was felt that the user should be warned of or be prevented from invoking transformation operations that would cause a state change into an illegal or undesireable state. In addition, it was felt that the end user should be warned as soon as possible about erroneous input. These goals lead to the decision to use LL(1) grammar for the syntax phase and for the inclusion of pre-transformation validation operations. LL(1) allows for the detection of errors at the earliest possible moment and allows for the calling of semantic action routines immediately following recognition of the first input token. Thus, a grammar rule might have the following form.

```
LHS : token1 {semantic1()} token2 {semantic2()} {semantic3()}
```

In the trivial case, semantic1() might be a pre-transformation operation that checks the legality of token1. Semantic2() might be a pre-transformation operation that checks the validity of token2 given the context of the current command and the context of token1. Semantic3() might be a transformation operation that should not be called unless all possible checks have been performed. The details of how to prevent the call of semantic3() in the face of semantic1() or semantic2() failure is presented in a later chapter.

Later in the research, it was decided that a good undo mechanism would reduce the need for protecting the transformation operations. If an error was made, the end user may easily roll the state of the system back to the prior state.

Design tools may operate in a structured or a non-structured mode. In a structured mode, every state entered is a legal one and the system will prevent all operations that will cause a transition into an illegal state. An experienced user will often

know of a short cut through a few illegal states that will eventually lead to a legal state. That path through uncharted territory may require far fewer steps than the path taken in the structured mode. In order to allow non-structured mode the pre-validation operations must be disabled when leaving structured mode and re-enabled when structured mode is re-entered. Since these types of operations need to be recognized by the tool-building system so that they may be enabled and disabled, they are identified separately in the specification. How to guarantee that the system is back in a legal state after a period in non-structured mode interaction is discussed in the following section covering time-invariant validation operations.

The following routines are typical of pre-transformation validation **ptvo** routines. Each ptvo returns an object or Nil.

```
ptvo ExistsInfluenceDiagram()
   --- Returns the Influence Diagram if it exists, Nil otherwise.
ptvo ExistsConcept(N: in name)
   --- Returns the named concept if it exists, Nil otherwise.
ptvo ExistsInfluence(affectC,causeC: in concept)
   --- Returns the influence connecting causeC and affectC
   --- if it exists, Nil otherwise.
```

4.5.4 Time-Invariant Validation Operations

Merely protecting against transitions into illegal model states is not enough. Due to a design oversight or to an excursion through non-structured mode, an illegal model state may be reached. It will be necessary to validate the legality of a model. A particularly important time to validate is when a design model is about to be stored in the design database.

The following set of routines may be sufficient to validate an influence diagram. Each time-invariant validation operation **tivo** returns an object or Nil.

```
tivo ValidID(ID: in influenceDiagram)
```

```
    --- Returns ID if it is a valid one, Nil otherwise.
    --- A valid diagram is one that passes the following predicates.
tivo ExistsOriginConcept(ID: in influenceDiagram)
    --- Returns Nil if there is not at least one concept with
    --- no cause influence, otherwise it returns a list of those
    --- concepts with no cause influence.
tivo ExistsFinalConcept(ID: in influenceDiagram)
    --- Returns Nil if there is not at least one concept with
    --- no affect influence, otherwise it returns a list of those
    --- concepts with no affect influence.
tivo NoOrphanConcepts(ID: in influenceDiagram)
    --- Returns Nil if there is no concept without a cause influence
    --- and without an affect influence, otherwise it returns
    --- a list of those concepts failing the test.
tivo NoDanglingInfluences(ID: in influenceDiagram)
    --- Returns Nil if there is no influence without a cause concept
    --- and without an affect concept, otherwise it returns
    --- a list of those influences failing the test.
tivo UnnamedConcepts(ID: in influenceDiagram)
    --- Each concept must have a name.
    --- Returns Nil if there is no concept without a name, otherwise
    --- it returns a list of unnamed concepts.
tivo UnnamedInfluence(ID: in influenceDiagram)
    --- Each influence must have a name.
    --- Returns Nil if there is no influence without a name, otherwise
    --- it returns a list of unnamed influences.
```

4.5.5 Trade-offs

It may not be immediately clear which set of operations are appropriate. Some designers prefer to identify the most primitive operations possible and then to build higher level operations out of them. If the operations are too primitive and are visible to the user, the interface semantics may be excessively complex. If the operations are too high level there is the risk of being unable to accomodate a variety of tool syntaxes, and the set of operations may be redundant to some degree.

For example, the tool designer may specify separate routines to add an influence to an affect concept and to add an influence to a cause concept.

```
to AddInfluenceToCauseConcept(C: in out concept; I: in out influence)
to AddInfluenceToAffectConcept(C: in out concept; I: in out influence)
```

The code for each would be simpler than the code for a single routine to add an influence to an affect and an influence concept. By specifying many small routines, there is less risk of miscommunication with the tool implementor. However, the two routine implementations will leave the model in a dangling state between their invocations. The wrong choice of transformation operations will complicate the user interface semantics as well as the pre-transformation validation operations. The question to ask is "What operations make sense to an end user?" and not "What operations make sense to a programmer?" A balance must be struck. The following one routine solution seems a reasonable compromise. Rather than specifying two operations, one is given, and the tool designer minimizes miscommunication with the tool implementor by using clear comments.

```
to AddInfluenceToConcepts
   (causeC,affectC: in out concept; I: in out influence)
   --- The influence, I, must exist.
   --- I must have neither causing nor affecting concepts.
   --- Each concept, causeC and affectC, must exist.
```

Whatever the choice, the tool designer must resist the temptation to over-specify the implementation and concentrate on the user interface semantics. The tool implementor must be able to decompose the operations in a way that allows for sharing and efficiency.

4.6 Conceptual Phase Compiler Outputs

Having discussed the conceptual specification of a sample tool, it is appropriate to turn our attention to the outputs of this phase's compiler. For completeness, the reader may wish to examine sections 4.10 and 4.11 where the complete syntax of the concept language and the complete IDE concept specification is given, respectively. Figure 4.4 is a simple data flow diagram of the OReO (Object, Relation, Operation) compiler.

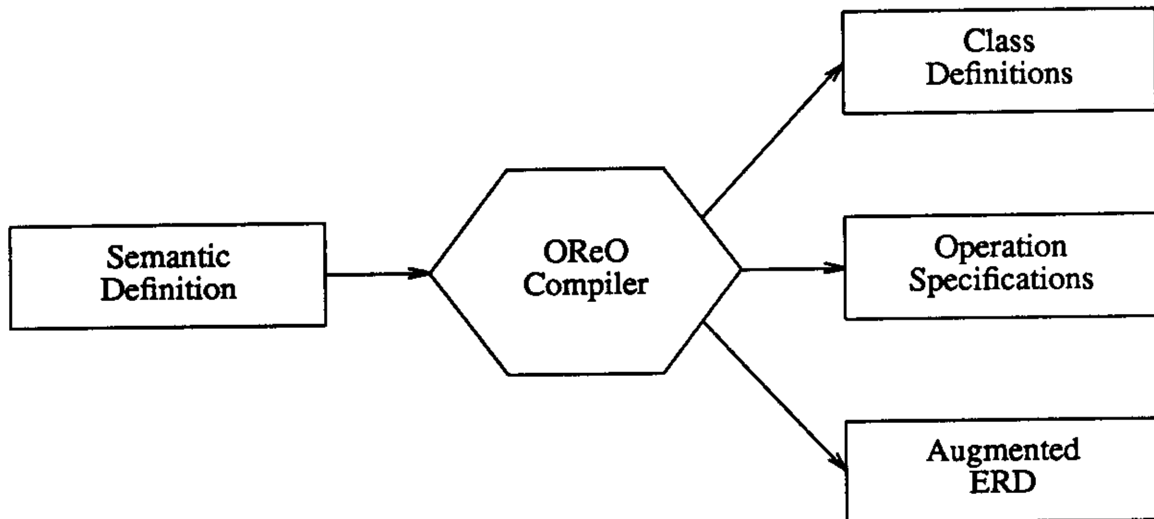


Figure 4.4: OReO Compiler

The first output of the compiler are the class definitions which incorporate the information extracted from the object and relation specification. They include the necessary data structures to represent an object, its attributes, and its relations with other objects. Routines to manipulate the relationships are also included.

The second output is the operation specification. This contains a specification of every operation identified in the operation step. Both the first and second outputs are provided to the tool implementor. The tool implementor must implement the specified operations by using the software provided from the class definitions.

The third output is an augmented Entity Relation Diagram, ERD, that is passed to an automated database management system compiler. This output is derived from the outputs of the object, attribute, and relation steps.

4.6.1 Class Definitions

Code can be generated directly from the various specification fragments in a rather obvious fashion. The prototype implementation of the SARA-UIMS is in the T language, a dialect of LISP. The following examples are in T. The objects for both influences and concepts are shown, but the object for the influence diagram is not.

Each T object is of the form

```
(define (MakeObjectName)
   (private part)
   (shared part))
```

This specification actually describes a class of objects. In order to make a new instance of this type of object, the function MakeObjectName is invoked. Each instance thus created will have its own copy of the private part but will share the contents of the shared part with all other objects of this class. In the following examples the components of the private part are *instance variables* and are specified as following.

```
(let (instance variable 1 definition)
    (instance variable 2 definition)
     ... )
```

The shared part is composed of *methods*.

```
(object nil
   (method 1)
   (method 2)
   ...
   (settable method 1)
   (settable method 2)
   ...
   (predicate method 1)
    ... )
```

The instance variables cannot be examined or set without going through one of the methods in the shared part. Methods are of several types,

1. simple methods that return a value, typically the value of an instance variable,

2. settable methods that typically set the value of one or more instance variables and perhaps return a value,

3. and predicate methods that typically return a boolean value.

Most of the T object is generated directly from the specification in an obvious way. However, every object of every type has some instance variables and shared routines that are not generated from their specifications. For example, each object has an instance variable to hold its graphics display data.

A T object is generated for each relation and each object identified in the conceptual definition phase. The T object that represents the influence relation and the concept object are shown following with their generating specification fragments preceding.

An influence has a proportionality. A causal concept influences many (j) affect concepts, and an affect concept is-influenced-by many (k) causal concepts.

```
(define (MakeInfluence)
 (let ((causalconcept-miv (MakeMiv "causal concept of influence"))
      (affectconcept-miv (MakeMiv "affect concept of influence"))
      (proportionality-miv (MakeMiv "proportionality of influence"))
      (semanticHelp-miv "An influence has a proportionality.
               A concept influences many (j) concepts,
               and a concept is influenced by many (k) concepts.")
      (graphics-miv (MakeMiv "graphics of influence")))
     (object nil
         ((draw self logicalDevice)
          ( ... ))
         ((causalconcept self) (cv causalconcept-miv))
         ((affectconcept self) (cv affectconcept-miv))
         ((graphics self) (cv graphics-miv))
         ((proportionality self) (cv proportionality-miv))
         ((semanticHelp self)
          (present semanticHelp-miv logicalDevice))
         (((setter causalconcept) self val)
          (set (cv causalconcept-miv) val))
         (((setter proportionality) self val)
          (set (cv proportionality-miv) val))
         (((setter affectconcept) self val)
          (set (cv affectconcept-miv) val))
         (((setter graphics) self val)
          (set (cv graphics-miv) val))
         ((hasGraphics? self) (not (null? graphics-miv)))
         ((influence? self) t))))
```

Figure 4.5: Influence Class Definition

A concept has a name.

An influence has a proportionality. A causal concept influences many (j) affect concepts, and an affect concept is-influenced-by many (k) causal concepts.

A concept (concepts) is an object. Initially, there are none; later, there may be many. Concepts may represent real-world events, human beliefs, or rather vague, non-quantifiable goals. Concepts influence each other.

```
(define (MakeConcept)
 (let ((name-miv (MakeMiv "name of concept"))
      (causalinfluences-miv (MakeMiv "causal influences of concept"))
      (affectinfluences-miv (MakeMiv "affect influences of concept"))
      (graphics-miv (MakeMiv "graphics of concept"))
      (semanticHelp-miv "The concept (concepts);
           initially there are none, later there may be many.
           Concepts may represent real-world events, human
           beliefs, or rather vague, non-quantifiable goals.
           A concept may have many (j) causal influences.
           A concept may have many (k) affect influences.
           A concept has a name.")
      (container-miv (MakeMiv "container of concept")))
     (object nil
         ((draw self logicalDevice)
           ( ... ))
         ((name self) (cv name-miv))
         ((causalinfluences self) (cv causalinfluences-miv))
         ((container self) (cv container-miv))
         ((affectinfluences self) (cv affectinfluences-miv))
         ((graphics self) (cv graphics-miv))
         (((setter name) self val) (set (cv name-miv) val))
         (((setter affectinfluences) self val)
          (set (cv affectinfluences-miv) val))
         (((setter causalinfluences) self val)
          (set (cv causalinfluences-miv) val))
         (((setter graphics) self val) (set (cv graphics-miv) val))
         ((hasGraphics? self) (not (null? graphics-miv)))
         ((concept? self) t))))
```

Figure 4.6: Concept Class Definition

The OReO compiler generates substantial portions of T code from the concept specification. Each object and each relation in the specification has a corresponding T object generated from it. The OReO compiler is responsible for generating the appropriate instance variables and methods for each T object.

Chapter 7 gives a full discussion of do, undo, and redo. However, the successful execution of do/undo/redo relies on the disciplined generation of T objects. Methods that alter the value of instance variables must be generated in a way that allows previous values of instance variables to be recalled.

The OReO compiler generates T objects according to the following rules. Instance variable names all end in "-miv". Miv is an acronym for Monitored Instance Variable. Mivs are data structures that contain a history of values rather than just a current value.

OBJECTS

A *specified object* is one that is specified in the object identification section.

1. Each specified object shall have a T object.

2. The name of an object's T object shall be the concatenation of "Make" and the string comprising the <subject> of the sentence introducing the object.

3. A semantic help instance variable, semanticHelp-miv, shall be generated and given a string value that is equivalent to the object identification paragraph.

4. Instance variables shall be generated from a relation identification sentence having a specified object as its <subject>. There shall be one instance variable generated for each <object> of such sentences. The instance variable name

shall be a concatenation of the attribute's name and ''-miv''.

5. Relation identification sentences having a specified object as <subject> shall be concatenated onto the object's semantic help instance variable.

6. Any relation identification paragraph that contains an independent clause having a specified object as its <subject> or <object> shall cause the generation of an instance variable. The instance variable's name shall be the concatenation of the object's modifier, the relation name (<verb>), and ''-miv''.

7. Each object that is the <object> of a relation sentence having an aggregating object as the <subject> shall have an instance variable, container-miv, generated.

8. An extraction method shall be generated for each instance variable. The name of the method shall be the same as the name of the instance variable without the ''-miv''. The method shall accept a single object as an argument and shall return the current value of the corresponding instance variable.

9. A setter method shall be generated for each instance variable. The name of the method shall be the same as the extraction method for the instance variable. The method shall accept both an object and a new value for the instance variable as arguments. The method shall set the current value of the instance variable to the second argument and shall return the updated object.

10. A predicate method shall be generated for each object. Its name shall be the concatenation of the object name and ''?''. The predicate shall return a true value if its argment is an object of this type.

11. Each object shall have generated methods that are required by SARA-UIMS.

These methods shall include, but not be limited to, a method to draw the object on a particular logical device and a method to provide semantic help.

RELATIONS

A *specified relation* is one that is introduced in the relation identification section.

1. Each specified relation shall have a T object.

2. The name of a relation's T object shall be the concatenation of ''Make'' and the string comprising the <subject> of the sentence introducing the relation.

3. A semantic help instance variable, semanticHelp-miv, shall be generated and given a string value that is equivalent to the relation identification paragraph.

4. Instance variables shall be generated from a relation identification topic sentence having a specified relation as its <subject>. There shall be one instance variable generated for each <object> of such sentences. The instance variable name shall be a concatenation of the attribute's name and ''-miv''.

5. Any relation identification paragraph that contains an independent clause having a specified object as its <subject> or <object> shall cause the generation of an instance variable. The instance variable's name shall be the concatenation of the object's modifier, the object's name, and ''-miv''.

6. An extraction method shall be generated for each instance variable. The name of the method shall be the same as the instance variable with the string ''-miv'' removed. The method shall accept a single relation as an argument and shall return the current value of the corresponding instance variable.

7. A setter method shall be generated for each instance variable. The name of the method shall be the same as the extraction method for the instance variable. The method shall accept both a relation and a new value for the instance variable as arguments. The method shall set the current value of the instance variable to the second argument and shall return the updated relation.

8. A predicate method shall be generated for each relation. Its name shall be the concatenation of the relation name and "?". The predicate shall return a true value if its argment is an object of this type.

9. Each relation shall have generated methods that are required by SARA-UIMS. These methods shall include, but not be limited to, a method to draw the relation on a particular logical device and a method to provide semantic help.

4.6.2 Operation Specification Output

This section describes the OReO compiler's output that is generated from the analysis of the operation section. This output is provided to the implementor as a template. The implementor is responsible for fleshing out the routine in accordance with its specification. The output could be identical to the input. However, it is possible to transform the operation specification into an executable *stub* in the implementation language, T.

Rather than describe the transformation process in algorithmic detail, simple, descriptive examples are provided. An example is provided for each of the operation types. Three minus signs (-) introduce a comment in the operation specification language. A semicolon introduces a comment in the T language. Comments terminate at the next end-of-line in both languages.

Transformation Operation

```
to ConceptNameSet(C: in out concept; N: in name)
  --- Returns Nil if it fails for any reason,
  ---    or C if it succeeds.
```

```
(define ConceptNameSet C N) ;;; a transformation operation
  ;;; C is a concept which is both input and output.
  ;;; N is a name which is input.
  ;;; Returns Nil if it fails for any reason,
  ;;;    or C if it succeeds.
```

Query Operation

```
qo Name(ID: in InfluenceDiagram; N: out name)
  --- Returns Nil if it fails for any reason,
  ---    or the N if it succeeds.
```

```
(define IDName ID N) ;;; a query operation
  ;;; ID is an InfluenceDiagram which is input.
  ;;; N is a name which is output.
  ;;; Returns Nil if it fails for any reason,
  ;;;    or N if it succeeds.
```

Pre-transformation Validation Operation

```
ptvo ExistsInfluence(affectC,causeC: in concept)
    --- Returns True if affectC influences causeC,
    ---    or Nil otherwise.
```

```
(define ExistsInfluence affectC causeC)
                    ;;; a pre-transformation validation operation
  ;;; affectC is a concept which is input.
  ;;; causeC is a concept which is input.
  ;;; Returns True if affectC influences causeC,
  ;;;    or Nil otherwise.
```

Time-invariant Validation Operation

```
tivo NoOrphanConcepts(ID: in influenceDiagram)
```

```
--- Returns True if there is no concept without a cause influence
--- and without an affect influence, otherwise it returns
--- a list of those concepts failing the test.


(define NoOrphanConcepts ID) ;;; time-invariant validation operation
   ;;; ID is an influenceDiagram which is input.
   ;;; Returns True if there is no concept without a cause influence
   ;;; and without an affect influence, otherwise it returns
   ;;; a list of those concepts failing the test.
```

4.6.3 Augmented ERD Output

The final output of the OReO compiler is the ERD corresponding to the specification. Relations are shown as diamonds, objects as rectangles, and attributes as ellipses. In the language of data base management, objects are entities, attributes are properties, and relations are relations. The unusual nature of the aggregating object, Influence Diagram, is clear by its containment of entities and by its lack of explicit connectivity. The lines connecting the entities and relations are labeled with the *role* played by the entity in the relation. The cardinality of the relation is shown by the lower case letters adjacent to the entity and the role connector. In the specification, the roles were j:1 and 1:k. When considered together by the database management system, they are folded together to form j:k. This is of interest only to the database management system. The interested reader is refered to Landis [Land86].

4.7 Vis-a-Vis IFIP WG 2.7 Reference Model

The IFIP WG 2.7 Reference Model [Beec85] provides a common vocabulary for discussing user interfaces and a reference point from which to compare implementations. Like the work presented in this dissertation, the reference model proposes a sequence of steps that take the reader and writer from concept towards

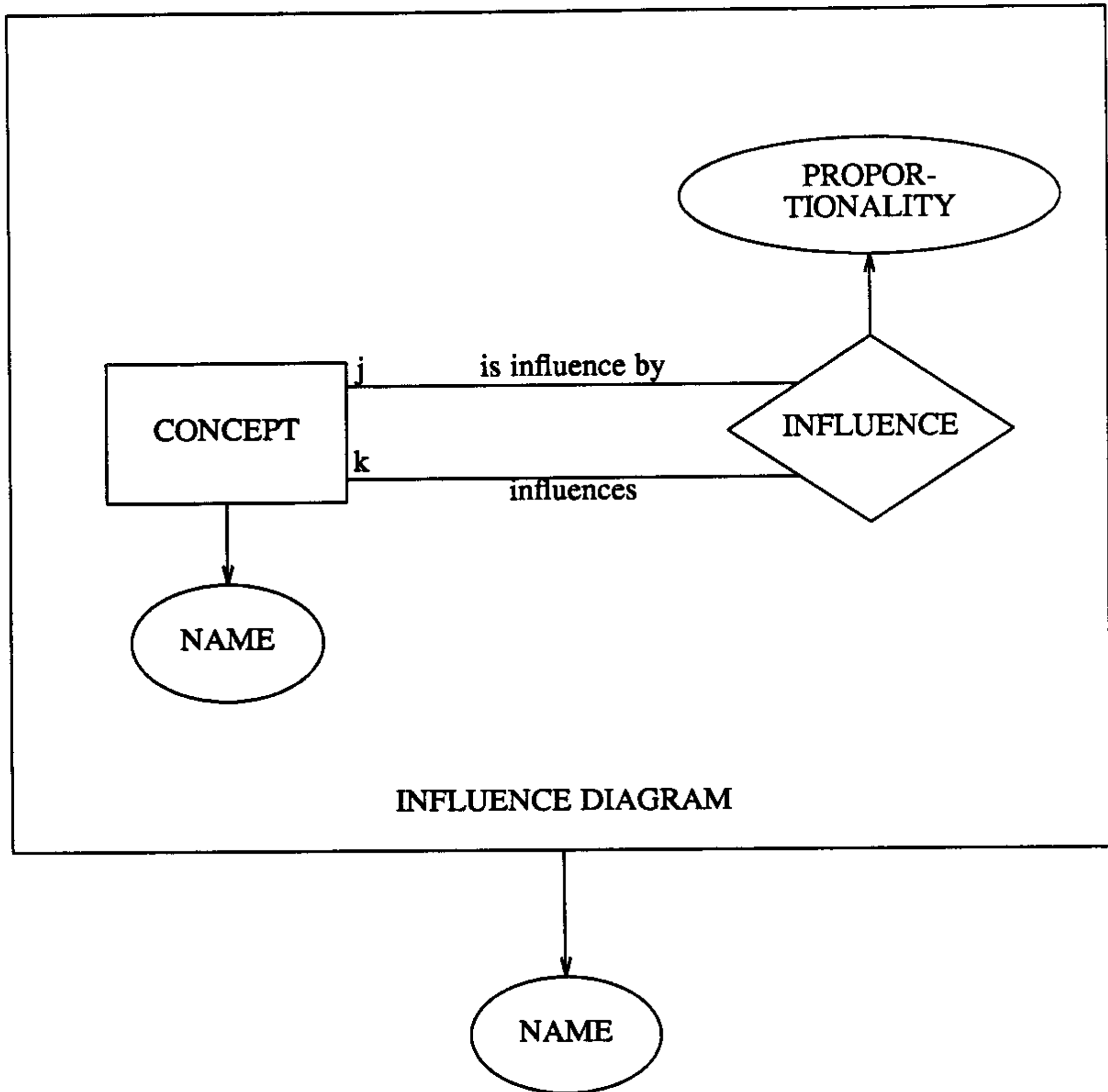


Figure 4.5: Entity Relation Diagram of Influence Diagram

implementation of the interface under discussion. Unlike the reference model, this work continues well into the implementation phase assuring that the final implementation coincides with its specification.

The steps of the reference model are roughly as follows.

For each object in the proposed user interface :

1. List the functional requirements of the object.
2. Use natural language to describe the object and its operations.
3. Objects and operations are described in subset of Affirm.
4. Complex semantics are optionally described in natural language.
5. Interfaces to operations are optionally described in VDM.

Implicit in the reference model is the initial step of identifying the objects visible at the user interface. Since one of the goals of this work is the analysis of the specification and fabrication of the specified interface, steps 2 and 4 of the reference model are supported by a language more rigorous than natural language and by a compiler for that language.

For purposes of comparison, the SARA-UI model is presented below.

Until a satisfactory description is obtained:
1. Identify all objects in the system.
2. Specify every relationship that exists between objects.
3. Specify all operations on objects.
    a. transformation operations,
    b. query operations,
    c. pre-transformation validation operations,
    d. time-invariant validation operations.

Each step has a formal language and language analyzer. The Until loop is supported by compiling the description and receiving information about the consistency of object use and operation application on objects. This support is not only provided within this and each other phase, but also between phases as they are successively integrated. Although the support software can offer assurances that the description is a valid one, the human tool designer has the final say in deciding when a satisfactory description is reached. The tool designer is able to take this specification to the potential end user population for review [Berr83]. This important source of feed back should not be ignored.

We cannot ignore the possible complexity of semantic operations and objects and the power of natural language. Each step, 1-3, allows optional natural language text to be provided along with each formal specification.

4.8 Vis-a-Vis Ada Packages

Ada packages have been used for design and for client review in much the same way the operation specifications are used here. Our operation specification language is very Ada-like. In fact, in the design phase of SARA/IDEAS we used Ada packages. They allow us to perform data abstraction and detailed design before beginning implementation. This procedure is described by Berry and Berry [Berr83]. Compiling the packages allowed us to verify the interface consistency of our design.

As implementation began in T the implementors were repeatedly faced with Ada's incompatibility with T in defining certain operations. For example, function names in T can contain a "?" and many predicate functions end in "?". Ada does not allow the "?" character in procedure names. The T language allows a variable number of arguments while Ada demands a fixed number.

All members of the group agreed that if the implementation language were Ada or an Algol-like language, Ada package specifications would be an excellent operation specification language, but given a LISP-like language for implementation, Ada packages were cumbersome.

4.9 Summary and Possible Improvements

It has been shown that the user interface semantics can be described in a natural way and that significant code generation is possible from the description. The semantic descriptions can be retained to be offered as help to the end user.

A companion dissertation [Land86] describes how the generated software and augmented Entity Relation Diagram can be input to a database compiler, resulting in even greater software generation and further integration of a new tool with the design environment.

The specification language has been developed sufficiently to allow description of design tools without much tolerance for free flowing natural language text. Although it is not implied that a full natural language interface should be provided, a great deal more freedom in sentence structure could easily be added.

Early in the research the relations were binary and quite simple, yet were sufficient to describe many of those relations found in design tools. The ability to describe ternary and more complex relations with near natural language was still an open research issue and was actively being researched [Chen83]. The paragraph structure introduced here allows for the description of extremely complex relations.

Design tools have been built using both structured and non-structured modes of operation. Each mode has strong proponents and opponents. The SARA tool-building system does not take the position of insisting on either mode of interaction. Rather, the mechanisms are provide to support each. The desire to provide mechanisms for support is pervasive in the SARA-UIMS. The enabling and disabling of pre-transformation validation operations is a good example of providing the mechanisms to support both modes without insisting on either.

With a good do/undo/redo facility, there is less need for pre-transformation validation operations. The recommendations from the next chapter on syntax specification concerning the choice of LL(1) grammars discuss this further.

4.10 The Concept Definition Language

The conceptual definition language and associated language processor is called the OReO language and OReO compiler, respectively. The name OReO brings attention to the importance of Objects, Relations, and Operations in this phase. Here is the YACC specification of the OReO language.

```
%{
#include "globals.h"

  /**************************************************************
   *            +++ Concept Parser +++                  *
   *                                       *
   **************************************************************/

%}
%union
   {
        int integer;
        char *string;
   };

                /*: Statement Key Words */
%token <unused>   AN
%token <unused>   A
%token <unused>   THE
%token <unused>   AND
%token <unused>   BUT
%token <unused>   OR
%token <unused>   NOR
%token <unused>   OBJECT
%token <unused>   RELATION
%token <unused>   ATTRIBUTE
%token <unused>   OPERATION
%token <unused>   IDENTIFICATION
%token <unused>   INITIALLY
%token <unused>   LATER
%token <unused>   THERE
%token <unused>   IS
%token <unused>   HAS
%token <unused>   HAVE
%token <unused>   ARE
%token <unused>   SHALL
%token <unused>   MAY
```

```
%token <unused>   MUST
%token <unused>   BE
%token <unused>   FROM
%token <unused>   ONE
%token <unused>   OF
%token <unused>   INTEGERTYPE
%token <unused>   CHARACTERTYPE
%token <unused>   FLOATTYPE
%token <unused>   STRINGTYPE
%token <unused>   TO
%token <unused>   QO
%token <unused>   PTVO
%token <unused>   TIVO
%token <unused>   IN
%token <unused>   OUT
%token <unused>   RETURNS

          /*: Brackets */
%token <unused>   LPAREN
%token <unused>   RPAREN
%token <unused>   LBRACE
%token <unused>   RBRACE

          /*: Separators */
%token <unused>   PERIOD
%token <unused>   COLON
%token <unused>   SEMICOLON
%token <unused>   COMMA
%token <unused>   PARAGRAPHSEPARATOR
          /*: Types */
%token <unused>   INTEGER
%token <unused>   LINGUISTICINTEGER
%token <unused>   WORD
%start semanticDescription
%%

semanticDescription : objectSection
                      relationSection
                      attributeSection
                      operationSection


/* OBJECT SECTION */


objectSection : objectSectionPreamble objects

objectSectionPreamble : OBJECT IDENTIFICATION PERIOD
                        PARAGRAPHSEPARATOR

objects : object
    | objects object
```

```
object : objectIntroSentence populationSentence optionalDescriptiveText
        PARAGRAPHSEPARATOR
objectIntroSentence : Subject optionalSynonym IS AN OBJECT PERIOD
populationSentence : objInitPart SEMICOLON objLaterPart PERIOD

optionalSynonym : /* empty */
        | parenedWord

objInitPart : INITIALLY COMMA stateOfBeingPhrase quantity

objLaterPart : LATER COMMA stateOfBeingPhrase quantity

optionalDescriptiveText : {TextMode(RandomText);}

objectName : name


/* RELATION SECTION */


relationSection : relationSectionPreamble relations
relationSectionPreamble : RELATION IDENTIFICATION PERIOD
                          PARAGRAPHSEPARATOR
relations : relation
        | relations relation
relation : relationParagraph PARAGRAPHSEPARATOR
relationParagraph : TopicSentence
        | TopicSentence roleSentences
roleSentences : roleSentence
        | roleSentences roleSentence
roleSentence : IndependentClause COMMA
        logicalConjunctive IndependentClause PERIOD
TopicSentence : Subject IS article RELATION PERIOD
        | SimpleSentence
        | SimpleSentenceWithCompoundObject
        | CompoundSentence
SimpleSentence : IndependentClause PERIOD
SimpleSentenceWithCompoundObject :
        Subject VerbPhrase NounPhraseList PERIOD
CompoundSentence : IndependentClause logicalConjunctive
        VerbPhrase Object PERIOD
        | IndependentClause COMMA DependentClauses PERIOD
IndependentClause : Subject VerbPhrase Object
DependentClauses : DependentClause
        | DependentClauses COMMA
        logicalConjunctive DependentClause
DependentClause : VerbPhrase Object
NounPhraseList : NounPhrase
        | NounPhraseList COMMA NounPhrase
NounPhrase : article Noun
Noun : attribute
    | simpleObject
```

```
    | aggregatingObject
simpleObject : name
aggregatingObject : name
relationName : name
roleName : name

Subject : NounPhrase
Object : article Noun
     | LINGUISTICINTEGER optionalSynonym Noun

VerbPhrase : IS
           | possessiveVerb
           | roleName


/* ATTRIBUTE SECTION */


attributeSection : attributeSectionPreamble attributes
attributeSectionPreamble : ATTRIBUTE IDENTIFICATION PERIOD
                                    PARAGRAPHSEPARATOR
attributes : attribute
             | attributes attribute
attribute : attributeSentence
          | attributeSentence PARAGRAPHSEPARATOR
attributeSentence  : article attributeName IS domain PERIOD
domain : article primitiveType
           | article set
           | FROM set
           | ONE OF set
attributeName : name


/* OPERATION SECTION */


operationSection : operationSectionPreamble operations
operationSectionPreamble : OPERATION IDENTIFICATION PERIOD
                           PARAGRAPHSEPARATOR {TextMode(Oper);}
operations : operation
           | operations operation
operation : transformationOperation
          | queryOperation
        | preTransformationValidationOperation
        | timeInvariantValidationOperation
operationName : name

transformationOperation :
   TO operationName argumentSpecification
queryOperation :
   QO operationName argumentSpecification RETURNS Noun
preTransformationValidationOperation :
```

```
   PTVO operationName argumentSpecification
timeInvariantValidationOperation :
   TIVO operationName argumentSpecification

argumentSpecification : LPAREN optionalArgSpecs RPAREN
optionalArgSpecs : /* empty */
                 | argSpecs
argSpecs : argSpec
         | argSpecs SEMICOLON argSpec
argSpec : formalArgs COLON argSide
formalArgs : formalArg
           | formalArgs COMMA formalArg
formalArg : name
argSide : type
        | IN type
        | OUT type
        | IN OUT type
        | OUT IN type
type : objectName
     | relationName
     | attributeName
     | aggregatingObject


/* COMMON NON-TERMINALS */


stateOfBeingPhrase : THERE beingVerb
                   | beingVerb article
beingVerb : IS
          | ARE
          | MAY BE
          | MUST BE
          | SHALL BE

possessiveVerb : HAS
               | MAY HAVE
               | MUST HAVE
               | SHALL HAVE

quantity : article | LINGUISTICINTEGER | INTEGER

parenedWord : LPAREN name RPAREN
set : LBRACE nameList RBRACE
nameList : name
         | nameList COMMA name

name : WORD
words : WORD
      | words WORD
primitiveType : INTEGERTYPE
              | CHARACTERTYPE
```

```
                | FLOATTYPE
                | STRINGTYPE
article : A | AN | THE
logicalConjunctive : AND | BUT | OR | NOR
%%
```

4.11 The Influence Diagram Editor Specification

Object Identification.

The influenceDiagram is an object. Initially, there are none; later, there is 1. Initially, the end user will not be provided with an empty influence diagram. It is the user's responsibility to create it. Only one influence diagram at a time will be allowed during Influence Diagram Editor use.

A concept (concepts) is an object. Initially, there are none; later, there may be many. Concepts may represent real-world events, human beliefs, or rather vague, non-quantifiable goals. Concepts influence each other.

Relation Identification.

The influenceDiagram has a name and has many (m) concepts.

A concept has a name.

An influence has a proportionality. A causal concept influences many (j) affect concepts, and an affect concept is-influenced-by many (k) causal concepts.

Attribute Identification.

A proportionality is one of [not,direct,inverse].
A name is a string.

Operation Identification.

to MakeInfluenceDiagram() **returns** influenceDiagram
--- A new, empty influence diagram will be created.
--- Only one influence diagram may exist at any time.
--- Returns Nil if it fails for any reason.
--- or a new influenceDiagram if it succeeds.
to MakeConcept() **returns** concept
--- A new, unnamed, disconnected concept will be created.
--- Returns Nil if it fails for any reason.
--- or a new concept if it succeeds.
to MakeInfluence() **returns** influence
--- A new, unnamed, disconnected influence will be created.
--- Returns Nil if it fails for any reason.
--- or a new influence if it succeeds.
to InfluenceDiagramNameSet(ID: **in out** influenceDiagram; N: **in** name)

```
   --- Returns Nil if it fails for any reason.
   --- If it succeeds, the name of I will be set to N and
   ---    the updated ID will be returned.
to ConceptNameSet(C: in out concept; N: in name)
   --- Returns Nil if it fails for any reason.
   --- If it succeeds, the name of C will be set to N and
   ---    the updated C will be returned.
to AddConceptToInfluenceDiagram
   (ID: in out influenceDiagram; C: in concept)
   --- Returns Nil if it fails for any reason.
   --- If it succeeds, C will be added to the collection of
   --- concepts contained in ID, the updated ID will be returned.
to AddInfluenceToConcepts
   (causeC,affectC: in out concept; I: in out influence)
   --- Returns Nil if it fails for any reason.
   --- Both concepts must exist, not already influence each other,
   --- and I must exist.
   --- If it succeeds, I will be added to the concepts and returned.
to ProportionInfuence(I: in out influence; P: in proportion)
   --- Returns Nil if it fails for any reason.
   --- I must exist. P must be a legitimate value.
   --- If it succeeds, P will be added to I and I will be returned.
to RemoveConceptFromInfluenceDiagram
   (ID: in out influenceDiagram; C: in concept)
   --- Returns Nil if it fails for any reason.
   --- ID must exist. C must exist.
   --- If it succeeds, C will be removed from the collection of
   --- concepts contained in ID, the updated ID will be returned.
to RemoveInfluenceFromConcept
   (affectC,causeC: in out concept; I: in out influence)
   --- Returns Nil if it fails for any reason.
   --- The concepts must exist. I must exist and connect the concepts
   --- as indicated.  If it succeeds, I will be disconnected from
   --- the concepts, and the updated influence will be returned.


qo conceptName(C: in concept) returns name
   --- Returns Nil if it fails for any reason,
   ---    or the name of C if it succeeds.
qo idName(ID: in InfluenceDiagram) returns name
   --- Returns Nil if it fails for any reason,
   ---    or the name of ID if it succeeds.


ptvo ExistsInfluenceDiagram()
   --- Returns the Influence Diagram if it exists, Nil otherwise.
ptvo ExistsConcept(N: in name)
   --- Returns the named concept if it exists, Nil otherwise.
ptvo ExistsInfluence(affectC,causeC: in concept)
   --- Returns the influence connecting causeC and affectC
```

```
    --- if it exists, Nil otherwise.


tivo ValidID(ID: In influenceDiagram)
    --- Returns ID if it is a valid one, Nil otherwise.
    --- A valid diagram is one that passes the following predicates.
tivo ExistsOriginConcept(ID: In influenceDiagram)
    --- Returns Nil if there is not at least one concept with
    --- no cause influence, otherwise it returns a list of those
    --- concepts with no cause influence.
tivo ExistsFinalConcept(ID: In influenceDiagram)
    --- Returns Nil if there is not at least one concept with
    --- no affect influence, otherwise it returns a list of those
    --- concepts with no affect influence.
tivo NoOrphanConcepts(ID: In influenceDiagram)
    --- Returns Nil if there is no concept without a cause influence
    --- and without an affect influence, otherwise it returns
    --- a list of those concepts failing the test.
tivo NoDanglingInfluences(ID: In influenceDiagram)
    --- Returns Nil if there is no influence without a cause concept
    --- and without an affect concept, otherwise it returns
    --- a list of those influences failing the test.
tivo UnnamedConcepts(ID: In influenceDiagram)
    --- Each concept must have a name.
    --- Returns Nil if there is no concept without a name, otherwise
    --- it returns a list of unnamed concepts.
tivo UnnamedInfluence(ID: In influenceDiagram)
    --- Each influence must have a name.
    --- Returns Nil if there is no influence without a name, otherwise
    --- it returns a list of unnamed influences.
```

CHAPTER 5

The Syntax Definition Phase

This chapter expands upon the notions initially presented in Chapter 3 and continues with the detailed exposition begun in Chapter 4. Fragments of the Influence Diagram Editor syntax specification are presented and refined to begin the discussion. Syntax rules defined in this phase are associated with semantic operations defined in the previous phase. In addition to the tool-specific operations, certain pre-defined system operations are introduced. The entire syntax specification language is given, and finally, the requirements of the syntax specification compiler are given.

The notation used for the syntax definition language should be familiar to any reader who has been exposed to formal language theory or to tools like Yet Another Compiler-Compiler, YACC [John75, John78].

5.1 Vocabulary and Introductory Discussion

Weaver [Weav68] offers the following definition of grammar and the distinction between grammar and syntax. ''Grammar may be defined as the science which describes the elements of language and the principles by which they are combined to form units of meaning. In some older studies of language, a distinction was often drawn between grammar and syntax. By this distinction, grammar was limited to the classification of elements and syntax to the rules which explain how they function together.'' In this dissertation the words grammar and syntax are used interchangeably.

Unfortunately for computer scientists, classification of a word (element) might be made based on the word's meaning, function, form, or some combination thereof. The state-of-the-art in efficient language processing insists that classification be unambiguous and be determined independently of the word's context. A limited amount of context sensitivity is available through the joint efforts of the lexical analyzer and parser.

Several other terms are used in the following discussion that deserve mention before proceeding. The terms *terminal* and *terminal symbol* are used to signify an indivisible unit in the language. Words and punctuation marks are examples of terminal symbols to an analyzer of natural language. The terms *non-terminal* and *non-terminal symbol* are used to describe an ordered collection of terminal symbols and/or non-terminal symbols. A sentence or a noun phrase is an example of a non-terminal to an analyzer of natural language. When the difference between terminals and non-terminals is unimportant they will collectively be called *tokens*.

During the syntax definition stage we are primarily concerned with the order of inputs, not how the input is to take place. A location may be input by moving a mouse until the cursor is on a desirable location and pushing a button. Alternatively, it may be input by a user typing (x,y) screen coordinates on a keyboard. We are concerned here with the ordering of tokens not interaction method or interaction device. Such logical or lexical considerations are the topic of the next chapter.

After specifying the syntax of all user commands, the syntax definer will attach semantic routines to them. The previous chapter introduced a variety of types of semantic operations that are available to the end user through the user interface, transformation, query, pre-transformation validation, and time-invariant validation operations. The syntax definer will select certain of those semantic operations to be invoked upon recognition of a syntactically correct command. The word *action* will be used when the type of semantic operation is unimportant.

A YACC specification has three sections separated by double percent signs.

```
TERMINAL SYMBOLS
%%
SYNTAX RULES
%%
USER DEFINED ACTIONS
```

The first section introduces the terminal symbols in the language that a separate lexical analyzer is expected to recognize and pass to the parser. The second section contains a description of the syntax rules to be recognized by the parser and calls to the actions to be made upon rule recognition. The third and final section contains actions written by the YACC user. These action routines implement the semantics of the language being defined.

YACC recognizes only two types of non-terminals, the single, distinguished non-terminal, or start state, and the remainder of the non-terminals. Whether used as an interpreter or as a compiler, the parser produced by YACC parses an entire sentential form derived from the distinguished non-terminal. If a programmer intends to build a compiler then the YACC specification will typically define a sentential form to be an entire computer program. The resultant YACC produced parser expects the input, up to the end-of-file marker, to be a sentential form. However, if the goal is to build an interpreter, then, typically, the YACC specification will define each legal statement or command to be a sentential form. The resultant YACC produced parser expects the entire input, up to the end-of-line marker (carriage return), to be a sentential form.

Rather than one sentential form, we wish to identify and parse many sentential forms that collectively comprise a dialogue. In a system with keyboard input, typically, each command is parsed separately, but no parsing is done until the carriage return is entered. In our system we wish to prompt, parse, error check, and offer help, as each token is input. In other words, we demand a great deal more interactivity than conventional compilers and interpreters afford.

Recall that an early defined requirement is that transformation operations are called only after all inputs are validated. In addition, the user view of a dialogue is that it consists of a sequence of command-response pairs. Because the parser needs to know when a transformation operation is legitimately placed and because the user gives special recognition to commands, the system provides three types of non-terminals. The distinguished non-terminal is a non-terminal associated with the tool system dialogue. A tool's dialogue is composed of commands and responses. A unique non-terminal is associated with each command. These non-terminals are

called sentence non-terminals. The other non-terminals are provided for reasons of abstraction and sharing.

Identification of the commands comprising a tool dialogue also happens to be a useful and convenient starting point in defining the Influence Diagram Editor tool syntax.

5.2 Syntax Specification

A SARA-TBS syntax specification has five sections. The five sections begin with %dialogue, %sentences, %rules, %terminals, and %shapes respectively.

```
%dialogue
DISTINGUISHED NON-TERMINAL
%sentences
NON-TERMINALS CORRESPONDING TO SENTENCES
%rules
SYNTAX RULES
%terminals
INPUT TERMINAL SYMBOL GROUPING
%shapes
OUTPUT SYNTAX OF TERMINAL SYMBOLS
```

Unlike YACC, the terminal symbol section comes last and plays a somewhat different role. The function of the lexical analyzer is provided by the logical device layer described in the next chapter. Also, there is no user-defined action section in the SARA-TBS syntax specification. The conceptual definition phase produces the specifications of the user-defined actions.

A first cut of the influence diagram editor syntax specification is given as follows.

```
%dialogue
    ideDialogue
%sentences
    initIDCmd
    addConceptCmd     delConceptCmd
```

```
    addInfluenceCmd   delInfluenceCmd
    redoCmd           undoCmd
    structuredModeCmd unStructuredModeCmd
    quitCmd

%rules
ideDialogue : initDesignCmd Command * quitCmd ;

initIDCmd : influenceDiagramName ;

Command : sysCommand
     | ideCommand
     ;

sysCommand : undoCmd
       | redoCmd
       ;

ideCommand : addConceptCmd
       | delConceptCmd
       | addInfluenceCmd
       | delInfluenceCmd
       | structuredModeCmd
       | unStructuredModeCmd
       ;
```

Each rule is composed of a left hand side, a colon, a right hand side, and a terminating semicolon. The left hand side is a single non-terminal that serves as a shorthand for the right hand side. The right hand side is either *simple* or *compound.* A simple right hand side is a sequence of tokens possibly including a *star,* "*", or *plus,* "+", meta character. The star implies that the previous token may be repeated zero or more times while the plus indicates one or more instances of the preceding token. A compound right hand side is simply a series of two or more simple right hand sides separated by an *or,* "|", meta character.

The single rule

```
aCommand : command1 | command2
```

is equivalent to the following pair of rules.

```
aCommand : command1
aCommand : command2
```

The next step in syntax specification is to expand each sentence non-terminal into an ordered sequence of terminal symbols and/or non-terminal symbols. The addConceptCmd might be specified verb-first.

```
addConceptCmd : addConcept centerPoint cName
```

This rule means that the token associated with the add concept command is followed by a token containing the screen location of the new concept which in turn is followed by the new concept's name.

The delete concept command might be specified in a verb-first fashion as

```
delConceptCmd : delConcept cPick
```

or object-first as

```
delConceptCmd : cPick + delConcept
```

which yields a more flexible command. The first variation says that the user is to provide an indication of the operation to be performed, delete concept, and is to pick the object it is to be performed upon, a concept. The second variation allows a user to pick one or more objects and terminate the list of objects by an indication of which operation is to be performed. It is a matter of taste as to whether verb-first or object-first is preferable, but it is a matter of fact that the parser can offer much more meaningful help if the verb-first variation is used. The context provided by the introductory verb allows the semantic help offered to the end user to be more direct.

A tool with purely keyboard input has the advantage that each command is terminated by a carriage return. A system with graphics input or mixed graphics and

keyboard input similarly needs to unambiguously determine when a complete command has been entered.

Object-first, verb-last commands accommodate this need easily. The lone verb in the command serves as verb and as command terminator. However, as noted previously, prompting and helping may be less definite.

Verb-first, object-last commands offer much better prompting and helping. However, end-of-command must be recognized unambiguously. End of command recognition is accomplished by sentence identification and by rule definition. Verb-first rules of the form

```
command : verb object *
```

do not give the command language interpreter the ability to detect end-of-command, EOC, until the verb of the following command is entered. Such rules are prohibited by the compiler and the syntax definer must provide an explicit terminator for the rule. For example,

```
command : verb object * EOC
```

where EOC is a token defined elsewhere. Current systems with a verb-first style and graphics input implement EOC as an out of bounds mouse click. For example, the end user might select a verb by a mouse click while the cursor is over a menu item and select many objects by repeatedly positioning the cursor over a desired object and clicking the mouse. The sequence is terminated by a mouse click when the cursor is positioned on an incompatible object or perhaps when the cursor is positioned out of the current window.

5.3 Augmenting Syntax with Semantic Actions

After all rules have been specified, the tool designer is ready to add calls to action routines. Recall that transformation operations can only be placed at the end of a sentence. The addConceptCmd might initially be augmented as follows.

```
addConceptCmd : addConcept cName centerPoint
        {
          (set C (MakeConcept))
          (ConceptNameSet C $2)
          (set (graphics C) $3)
          (AddConceptToInfluenceDiagram ID C)
        }
       ;
```

Although this first cut is not totally correct, it captures the designer's intent well enough to proceed. The simplified T code specifies that a new concept, C, is created; its name is set to the value of the cName token, $2; its graphics part is given the value of the centerPoint token, $3, which might be the center point; and the new concept will be added to the influence diagram, ID, which at this point is assumed to be a global variable.

A more robust implementation might include validation operations embedded within the sentence rather than just a transformation operation at the end of the sentence.

```
addConceptCmd : addConcept
          cName {(dt(eq? (ExistsConcept($2)) nil))}
          centerPoint
        {
          (set C (MakeConcept))
          (ConceptNameSet C $2)
          (set (graphics C) $3)
          (AddConceptToInfluenceDiagram ID C)
        }
       ;
```

The ExistsConcept ptvo will pass or fail. If it fails, the discard token system

operation, dt(), will cause the previous token to be discarded and the parser will continue as if the discarded token had not been input. A related system operation, ds(), discards all portions of the current sentence being input.

The full power of the implementation language is available between the braces, {}, so that arbitrarily complex expressions can be tested for the dt() and ds() operations and so that user defined operations can be called conditionally or repeatedly.

5.4 Grouping Terminal Symbols

The next section of the syntax specification is the %terminals section. It arguably belongs in the logical device specification, but is included here because its inclusion allows the grammar compiler to pass a minimum of information to the next phase, the device compiler. It also allows the compiler to check that each token is either a terminal or a non-terminal. If a token appears on the left hand side of a rule, then it is a non-terminal. If a token is identified in the %terminals section, then it is a terminal symbol that is directly input by the end user. The appearance of a token in the grammar that is not identified as either a non-terminal or a terminal signifies an error in the grammar. The process of grouping terminal symbols comes late in syntax specification because it requires the syntax definer to shift from thinking about purely syntactic issues into thinking about lexical issues.

Eventually, all terminal symbols will be input through some logical device and ultimately through a physical device. For uniformity, it is desirable that similar terminal symbols be prompted for, and be input in a similar fashion.

Terminal symbols, i.e., those that did not appear as the left hand side of a rule, are associated with similar terminals. Such associations, or bindings, are specified by a *binding rule*. A binding rule has a left hand side, a colon, and a right hand side. The left hand side is a single word and the right hand side is a simple list of terminal symbols.

```
%terminals

verbs :
   structuredMode unstructuredMode
   addConcept    delConcept
   addInfluence  delInfluence
   quit

points :
   centerPoint

entities :
   concept influence influenceDiagram

strings :
   conceptName influenceProportion influenceDiagramName
```

The verb-first style of syntax that has been used in this chapter has produced a %rules section with a verb-like token as the first symbol on the right hand side of each sentence. Each verb may be prompted for by "enter verb" requiring the end user to respond via a keyboard or verbs may be prompted for by displaying a menu requiring the end user to select one choice by a mouse click. In any case, it is reasonable that all verbs will be input in the same fashion.

A more complete discussion of the use of the information extracted from the %terminals section is presented in the following chapter.

5.5 Output Terminal Symbols

The last section is the %icons section. Like the %terminals section, the %icons section could be placed in the logical device specification. The %terminals section groups tokens and identifies them as input terminal symbols. The %icons section assigns a display shape to each object that will be displayed on a logical output device. The %icons and %terminals sections collectively represent a shift from input syntax to lexical input and lexical output.

The assignment of shapes to terminal symbols is used to extract drawing routines from system libraries. The words appearing on the left hand side of assignment rules are not limited to geometric shapes such as circles and rectangles. The only requirement is that the system libraries contain support software for the display and selection of the shape indicated on the left hand side of the rule. A menu is an obvious candidate for such library support.

```
%icons
   rectangle : influenceDiagram ;
   circle : concept ;
   polyline : influence ;
   text : conceptName influenceProportion influenceDiagramName ;
```

5.6 Aggregating Tool Syntaxes

Before proceeding with the influence diagram editor syntax specification, recall that the influence diagram editor is anticipated to be just one tool amongst many in a system of tools. We might consider the entire system to be a separate tool with a syntax composed of sub-dialogues. That top level tool could have its syntax specification as following.

```
%dialogue
   topLevelToolDialogue
%sentences
```

```
    undoCmd
    redoCmd
    quitCmd
    flowChartEditorCmd
    ideCmd
%rules
    topLevelToolDialogue : Command * quitCmd ;
    Command : flowChartEditorCmd flowChartEditorDialogue
            | ideCmd ideDialogue
            | undoCmd {(undo)}
            | redoCmd {(redo)}
            ;
```

It should be clear at this point that the non-terminal introduced in the %dialogue section of the influence diagram editor is only a *relatively distinguished* non-terminal. It appears to be just another non-terminal to the top-level tool.

Each sub-dialogue such as ideDialogue forms an encapsulation of the name space of that tool's dialogue. Only the %dialogue non-terminal is known outside of that encapsulation. This prevents conflicts between the names of non-terminals in different tools. To support sharing of common non-terminals, undoCmd and redoCmd for example, the parser will first look within a sub-dialogue for a non-terminal and will look upward to the containing dialogues for unfound non-terminals. In order to guide the name resolution algorithm, the %dialogue section also allows statements of the following form.

```
%dialogue
    ideDialogue is a sub-dialogue of topLevelToolDialogue
```

5.7 Influence Diagram Editor Syntax

This section contains a complete, possible syntax specification for the influence diagram editor.

```
%dialogue
    ideDialogue is a sub-dialogue of topLevelToolDialogue
```

```
%sentences
   initIDCmd
   addConceptCmd
   delConceptCmd
   addInfluenceCmd
   delInfluenceCmd
   structuredModeCmd
   unStructuredModeCmd
   quitCmd

%rules
ideDialogue : initDesignCmd Command * quitCmd ;

initIDCmd : influenceDiagramName
        {
          (lset ID (MakeInfluenceDiagram))
            ;;; define ID to be global
          (InfluenceDiagramNameSet ID $1)
            ;;; and set its name
        }
        ;

Command : sysCommand
       | ideCommand
       ;

sysCommand : undoCmd
        | redoCmd
        ;

ideCommand : addConceptCmd
        | delConceptCmd
        | addInfluenceCmd
        | delInfluenceCmd
        | structuredModeCmd
        | unStructuredModeCmd
        ;

structuredModeCmd : structuredMode
                {
                (cond ((ValidID ID) (ptvo enable))
                    (else (print invalid influence diagram)))
             }
      ;

unStructuredModeCmd : unStructuredMode { (ptvo disable) }
      ;

addConceptCmd : addConcept cName cLocation
           {
             (let ((C (MakeInfluenceDiagram)))
                (set C (MakeConcept))
```

```
                (ConceptNameSet C $2)
                (set (graphics C) $3)
                (AddConceptToInfluenceDiagram ID C))
          }
        ;

delConceptCmd : delConcept cPick
          { (RemoveConceptFromInfluenceDiagram ID $2) }
        ;

addInfluenceCmd : addInfluence cPick iProportion cPick
            {
              (let ((I (MakeInfluence)))
                 (ProportionInfluence I $3)
                 (AddInfluenceToConcept $2 $4)
                 (RouteInfluence from $2 to $4))
                  ;;;operation not defined.
            }
          ;

delInfluenceCmd : delInfluence iPick
          {
             (RemoveInfluenceFromConcept
              (causalconcept $2)
              (affectconcept $2) $2)
             (DiscardInfluence $2) ;;;operation not defined.
          }
        ;

quitCmd : quit ;

cPick : concept ;
iPick : influence ;
cLocation : centerPoint ;
cName : conceptName ;
```

%terminals

```
verbs :
   structuredMode unstructuredMode
   addConcept    delConcept
   addInfluence   delInfluence
   quit ;

points :
   centerPoint ;

entities :
   concept influence influenceDiagram ;

strings :
   conceptName influenceProportion influenceDiagramName ;
```

%icons

```
rectangle : influenceDiagram ;
circle : concept ;
polyline : influence ;
text : conceptName influenceProportion influenceDiagramName ;
```

During this phase of specification, the syntax definer is focusing on end user input commands and on the tool's response to those commands. While attempting to describe this dialogue, the designer may become aware that certain operations are required for the dialogue that were not included in the output from the conceptual definition phase.

The actions RouteInfluence and DiscardInfluence were not part of the conceptual definition. Any omission or error detected in this phase is fed back into a refinement of the first phase.

5.8 The Syntax Specification Compiler

The syntax definition phase logically falls between the semantic definition phase and lexical (logical device and physical device) phase. The syntax or grammar compiler accepts inputs from the compiler supporting the semantic definition phase and produces outputs for use by the lexical phase.

Recall the diagrams of the respective compilers first presented in Chapter 3.

Operation specifications are provided as input so that the grammar compiler can compare the specifications with their actual use in the %rules section. Successful comparison implies that only defined operations are used, that all defined operations are used, and that operations are invoked with the correct type and number of arguments. A discussion of optional approaches is included in the conclusion section of

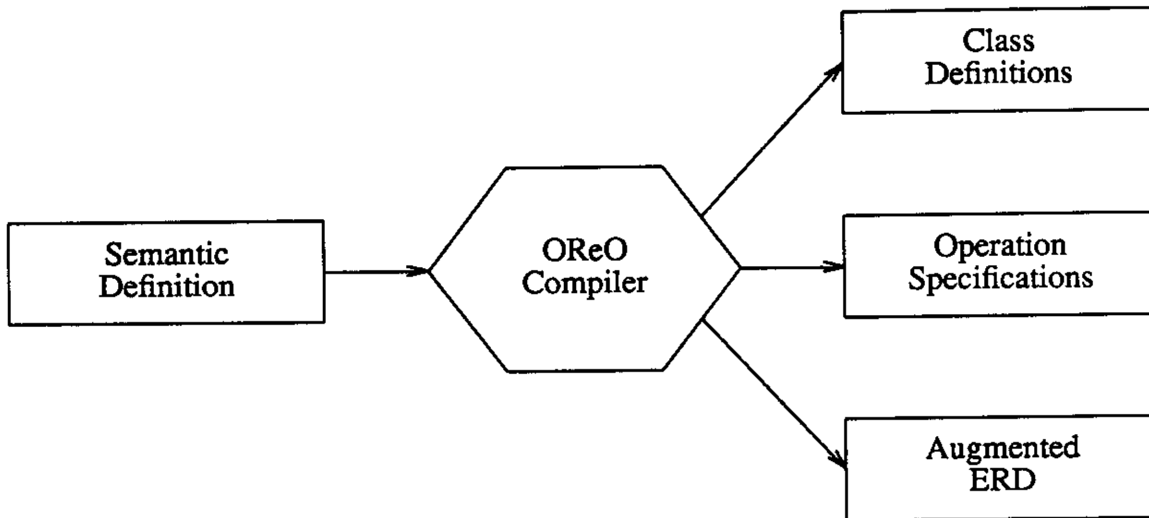


Figure 5.1: OReO Compiler

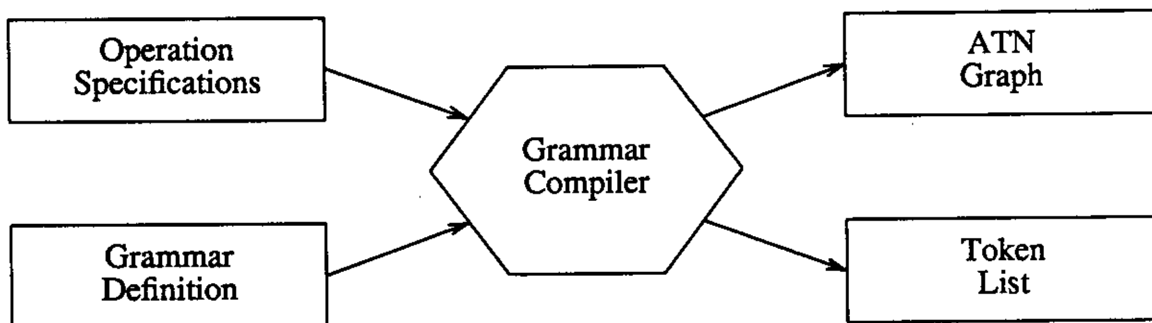


Figure 5.2: Grammar Compiler

this chapter.

The grammar definition input to the compiler has been thoroughly presented in previous sections. The compiler is responsible for assuring the LL(1) property of the input grammar. Again, the conclusion section of this chapter discusses alternatives and relevant experience gained from construction of the prototype compiler.

The remainder of this section is devoted to describing the two outputs of the grammar compiler, the ATN graph and the token list.

5.8.1 Augmented Transition Network Output

In addition to carrying out the algorithms necessary to assure the LL(1) property [Lewi76a], the grammar compiler is responsible for generating some common form of the language that eventually guides the execution of the command-response language interpreter. Several common forms were considered during the research including parse tables, a collection of procedures that implement recursive descent parsing, the original grammar reorganized into a LISP list structure, and finite state transition networks. Not only did the selected common form have to be sufficient to guide the interaction, it had to support syntactic and semantic help queries and to provide a linkage between syntax recognition and semantic action.

The augmented transition network, ATN [Wood70], was selected because it best met all of those requirements and because the ATN representation is highly readable by humans. Woods states that, ''If the transition network model were implemented on a computer with a graphics facility for displaying the network, it would be one of the most perspicuous (as well as powerful) grammar models available.''

In the power spectrum from context-free grammars [Lewi76b] to context-sensitive grammars to transformational grammars [Chom63], Woods claims that the transition network model has the full power of transformational grammars with execution costs only marginally greater than that for context free grammars.

The ATN, as Woods proposes it, has nodes which represent states and directed arcs connecting the nodes. An ATN has a distinguished start state, a set of accept states, and a set of intermediate states. If input is totally consumed while the execution mechanism is in one of the accept states then the input is a legal sentential form in the language. Arcs are labeled with terminal or non-terminal symbols.

If, for example, we wish to recognize simple noun phrases of the following form

Rover

the big black dog

the dog

we can describe those phrases syntactically as follows.

```
nounPhrase : article adjective * noun
        | properNoun
```

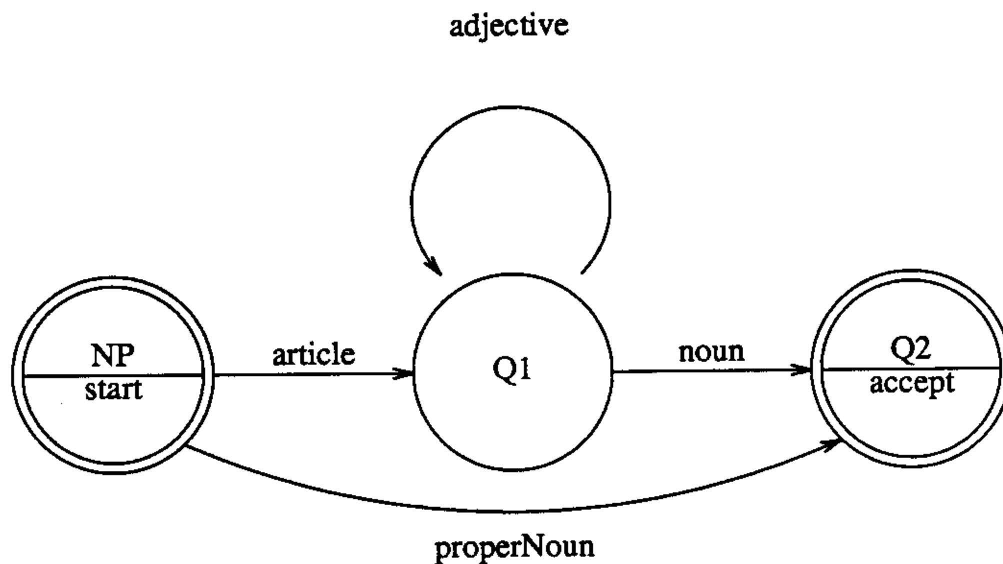


Figure 5.3: Noun Phrase ATN

In the start state, NP, the input of an article causes the execution mechanism to transition into state Q1, while the input of a proper noun causes an input into state Q2, the only accept state in this graph.

The ATN has state variables called *registers* that are used to record any relevant history. For example, when encountering the pronoun ''it'' it may be necessary to recall the antecedent from earlier in the sentence or even from a preceding

paragraph.

The arcs of the transition network are augmented by *conditions* and *actions*.

Unlike a finite state transition network, the transition from NP to Q1 is taken if and only if the next input is an article and the arc's augmenting condition is met. Conditions can be as simple as *true*, the transition is guided by the next input only, or can be more complex containing an arbitrarily complex logical expression with references to registers.

Arcs are also augmented with actions. When a transition is taken, the action attached to the associated transitional arc is executed. A common use of actions is to compute and set registers.

Part of the top level tool dialogue is shown below and is used as the basis for the ensuing exposition.

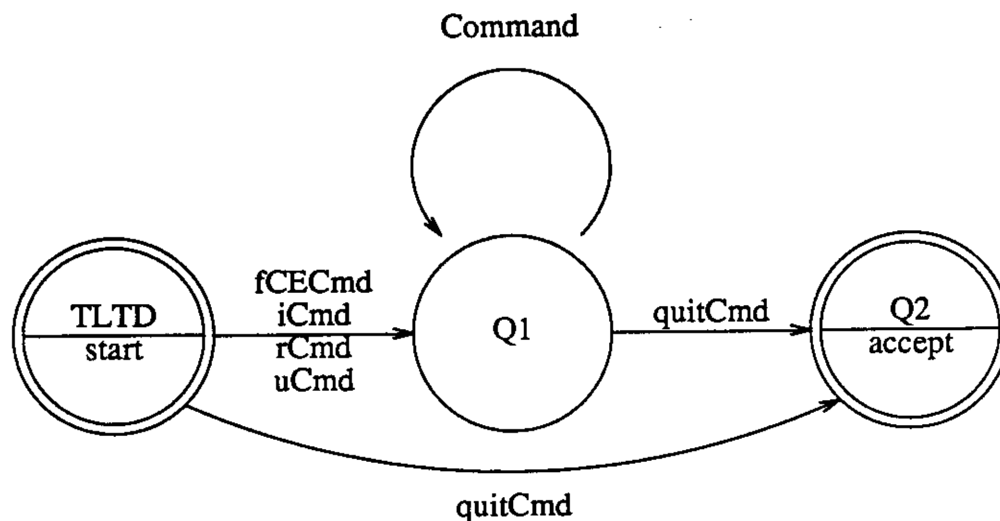


Figure 5.4: Top Level Tool Dialogue ATN

Rather than generating a single, large graph, the ATN for a dialogue is composed of a collection of disjoint graphs, one per non-terminal including the dialogue non-terminal. Outgoing arcs are augmented with a *first* register containing the terminal symbols that can begin the arc's non-terminal. LL(1) guarantees that any legal input terminal will be an element of exactly one first set.

It is conceivable and perhaps desireable that instead of writing a linear representation of the syntax of a tool, a tool with graphical interaction could be offered to the syntax definer. This tool could allow the direct expression of ATNs in a graphical form and perform the LL(1) analysis upon user request (invocation of a ValidATN time-invariant validation operation). If it is true that ATN construction could be supported by an interactive, graphical tool, then the conceptual definition language presented in the last chapter should be suitable for describing the modified ATN that is generated by the SARA-TBS grammar compiler.

Object Identification.

The ATN is an object. Initially, there are none; later, there is 1. Initially, the end user will not be provided with an empty ATN. It is the user's responsibility to create one for the dialogue non-terminal. Only one ATN is constructed at a time.

A graph (graphs) is an object. Initially, there are none; later, there may be many. There will be one graph for the dialogue non-terminal, and there will be one graph for each remaining non-terminal in the dialogue. A graph is made up of nodes and transitions between nodes.

A node (nodes) is an object. Initially, there are none; later, there may be many. Nodes can be start states, intermediate states, or accept states.

A token (tokens) is an object. Initially, there are none; later, there may be many. Tokens are terminal or non-terminal symbols. Terminal symbols are indivisible units in the language and are produced by the lexical level. Non-terminal symbols are divisible units in the language and are produced by collecting and ordering terminals and non-terminals.

A graphCallStack (GCS) is an object. Initially, there is 1; later, there is 1. An ATN will have one GCS. During parsing a non-terminal may be encountered and something similar to a subroutine call (sub-graph call) will be made to the graph associated with the non-terminal. The GCS will support subgraph calls.

Relation Identification.

The ATN has a name, has a graphCallingStack, and has many graphs.

A graph has a name and has many nodes.

A node has a nodeType and has many transitions.

A token has a name and has a tokenType.

A transition has a token and has a firstRegister. A source node transitions-to a destination node.

Attribute Identification.

A nodeType is one of [start,intermediate,accept].

A tokenType is one of [terminal,non-terminal,unknown].

A name is a string.

A firstRegister is a list of tokens.

Operation Identification.

--- Transformation Operations.

to MakeATN() **returns** ATN
--- A new, empty ATN will be created.
--- Only one ATN may exist at any time.
--- Returns Nil if it fails for any reason.
--- If it succeeds, the new ATN will be returned.

to MakeGraph() **returns** graph
--- A new, unnamed, disconnected graph will be created.
--- It will contain an empty subgraph call stack.
--- Returns Nil if it fails for any reason.
--- If it succeeds, the new graph will be returned.

to MakeNode() **returns** node
--- A new, empty, disconnected node will be created.
--- Returns Nil if it fails for any reason.
--- If it succeeds, the new node will be returned.

to MakeToken() **returns** token
--- A new, unnamed, token will be created.

--- Returns Nil if it fails for any reason.
--- If it succeeds, the new token will be returned.

to MakeTransition() **returns** transition
--- A new, disconnected transition will be created.
--- Returns Nil if it fails for any reason.
--- If it succeeds, the new transition will be returned.

to ATNNameSet (A: **in out** ATN; N: **in** name)
--- Returns Nil if it fails for any reason.
--- If it succeeds, the name of A will be set to N and
--- the updated A will be returned.

to AddGraphToATN (G: **in** graph; A: **in out** ATN)
--- Returns Nil if it fails for any reason.
--- If it succeeds, G will be added to the ATN and
--- the updated ATN will be returned.

to RemoveGraphFromATN (G: **in** graph; A: **in out** ATN)
--- Returns Nil if it fails for any reason.
--- If it succeeds, G will be removed from the ATN and
--- the updated ATN will be returned.

to GraphNameSet (G: **in out** graph; N: **in** name)
--- Returns Nil if it fails for any reason.
--- If it succeeds, the name of G will be set to N and
--- the updated G will be returned.

to AddNodeToGraph (N: **in** node; G: **in out** graph)
--- Returns Nil if it fails for any reason.
--- If it succeeds, N will be added to the collection of
--- nodes contained in G, the updated G will be returned.

to RemoveNodeFromGraph (N: **in** node; G: **in out** graph)
--- Returns Nil if it fails for any reason.
--- If it succeeds, N will be removed from the collection of
--- nodes contained in G, the updated G will be returned.

to NodeTypeSet (N: **in out** node; T: **in** nodeType)
--- Returns Nil if it fails for any reason.
--- If it succeeds, the node type of N will be set to T
--- and the updated N will be returned.

to AddTransitionToNodes (T: **in out** transition; sN,dN: **in out** node)
--- sN and dN are source and destination node, respectively.
--- Returns Nil if it fails for any reason.
--- If it succeeds, T will be added to the collection of
--- transitions out of sN and into dN. The updated sN, dN,
--- and T will be returned.

to RemoveTransitionFromNodes (T: **in out** transition; sN,dN: **in out** node)
--- sN and dN are source and destination node, respectively.

--- Returns Nil if it fails for any reason.
--- If it succeeds, T will be removed from the collection of
--- transitions out of sN and into dN. The updated sN, dN, and T
--- will be returned.

to TransitionTokenSet (Tr: **in out** transition; Tok: **in** token)
--- Returns Nil if it fails for any reason.
--- If it succeeds, Tok will be assigned to Tr and
--- the updated Tr will be returned.

to TransitionFirstRegisterSet (Tr: **in out** transition; FR: **in** firstRegister)
--- Returns Nil if it fails for any reason.
--- If it succeeds, FR will be assigned to Tr and
--- the updated Tr will be returned.

to PushGraphOnGCS (G: **in** graph; Gcs: **in out** GCS)
--- Returns Nil if it fails for any reason.
--- If it succeeds, G will become the top of the Gcs and
--- the updated Gcs will be returned.

to PopGraphFromGCS (G: out graph; Gcs: **in out** GCS)
--- Returns Nil if it fails for any reason.
--- If it succeeds, the top element of Gcs will be removed and
--- and the top element of the Gcs will be
--- assigned to G, the updated Gcs will be returned.

to ForwardStateChange (A: **in out** ATN; N: **in out** node; T: **in** token)
--- Returns Nil if it fails for any reason.
--- T will be searced for in the first register of each of N's
--- transitions. The destination node of the matching transition
--- will become the new value of N. The old value of N will be
--- pushed onto the GCS of A. Any actions associated with the
--- transition will be executed.

to BackwardStateChange (A: **in out** ATN; N: **out** node)
--- Returns Nil if it fails for any reason.
--- N will be set to the top element of A's GCS. The top of
--- A's GCS will be discarded.
--- transition will be executed.

--- Query Operations.

qo ATNName(A: **in** node) **returns** name
--- Returns Nil if it fails for any reason,
--- or the name of A if it succeeds.

qo graphName(G: **in** graph) **returns** name
--- Returns Nil if it fails for any reason,
--- or the name of G if it succeeds.

qo tokenName(T: **in** token) **returns** name
--- Returns Nil if it fails for any reason,

--- or the name of T if it succeeds.

qo TopOfGCS (Gcs: **in** GCS) **returns** graph
--- Returns Nil if it fails for any reason.
--- If it succeeds, the top element of Gcs will be returned

qo typeOfToken (T: **in** token) **returns** tokenType
--- Returns Nil if it fails for any reason.
--- If it succeeds, the token type of T will be returned

qo sourceOfTransition (T: **in** transition) **returns** node
--- Returns Nil if it fails for any reason.
--- If it succeeds, the source node of T will be returned

qo destinationOfTransition (T: **in** transition) **returns** node
--- Returns Nil if it fails for any reason.
--- If it succeeds, the destination node of T will be returned

--- Pre-transformation Validation Operations.

ptvo ExistsATN()
--- Returns the ATN if it exists, Nil otherwise.

ptvo ExistsGraph(N: **in** name)
--- Returns the named graph if it exists, Nil otherwise.

ptvo ExistsToken(N: **in** name)
--- Returns the named token if it exists, Nil otherwise.

ptvo TokenInFirst(T: **in** token; FR: firstRegister)
--- Returns TRUE if T is in FR, Nil otherwise.

--- Time Invariant Validation Operations.

tivo ValidATN(ATN: **in** ATN)
--- Returns ATN if it is a valid one, Nil otherwise.
--- A valid diagram is one that passes the following predicates.

tivo ExistsUniqueDialogueNode(ATN: **in** ATN)
--- Returns the graph associated with the dialogue non-terminal,
--- Nil otherwise. There must be only one such graph.

tivo ExistsInfiniteGraphs(ATN: **in** ATN)
--- Returns Nil if there is no graph without at least one
--- accept node, otherwise it returns a list of those
--- graphs with no accept node.

A complete discussion of how the command-response language interpreter traverses and manipulates an ATN is given in chapter 7.

5.8.2 Token List Output

The grammar compiler collects, categorizes, and outputs the symbols that appear in the syntax specification. The collection and categorization of symbols can most easily be understood as a single pass over the syntax specification. Those tokens found on the left hand side of a rule in the %rules section are categorized as *local non-terminals*. Those tokens found on the right hand side of a rule in the %terminals section are categorized as *terminal symbols*. Those tokens found on the right hand side of a rule in the %rules section, but not categorized as terminal symbols or local non-terminal symbols, are suspected of being *external non-terminals*. If the %dialogue section specifies a parent dialogue then the token list for the parent dialogue is searched for those suspected external non-terminals. If the search fails to produce a match, or if the %dialogue section does not specify a parent dialogue, then the remaining suspected external non-terminals are errors.

The token list is thus a partitioned list composed of four partitions.

The first partition is the dialogue non-terminal part, which contains the non-terminal associated with the non-terminal specified in the %dialogue section of the specification being compiled. This partition is the only partition visible to those dialogues not claiming to be a sub-dialogue. In addition it is only visible to the parent dialogue that is specified in the %dialogue section.

The second partition is the local non-terminal part. It contains a list of the local non-terminals introduced in the current specification. This part is visible only to those dialogues that claim this dialogue as parent.

The third partition contains a list of terminal symbol bindings. This list is itself composed of lists. Each binding rule produces one small list with the left hand side of the binding rule as the first element and the elements of the right hand side of the rule comprising the remainder of the list. The third partition, the terminal symbol binding partition, is an unordered list of these smaller lists. This partition is only of interest to the device compiler.

The fourth and final partition is the list of unresolved tokens. These tokens are of interest to the syntax definer.

The token list was prototyped as a T object with an instance variable for each partition and a method for inserting a new element in each partition. The insertion methods guaranteed that only one instance of any token would be found in a partition. Methods were also provided to print the entire token list or any single partition.

The list of symbols in the grammar will be computed and output for use by the device compiler (see chapter 6).

5.9 Conclusions and Possible Improvements

5.9.1 Alternative Syntax Experiment

More than one syntax can be built using the same operations. This procedure was followed to produce different syntaxes for the Structure Model Tool of SARA-IDEAS, and in fact, was used to provide a single, uniform syntax for three different implementations of the Structure Model Tool [Cai85].

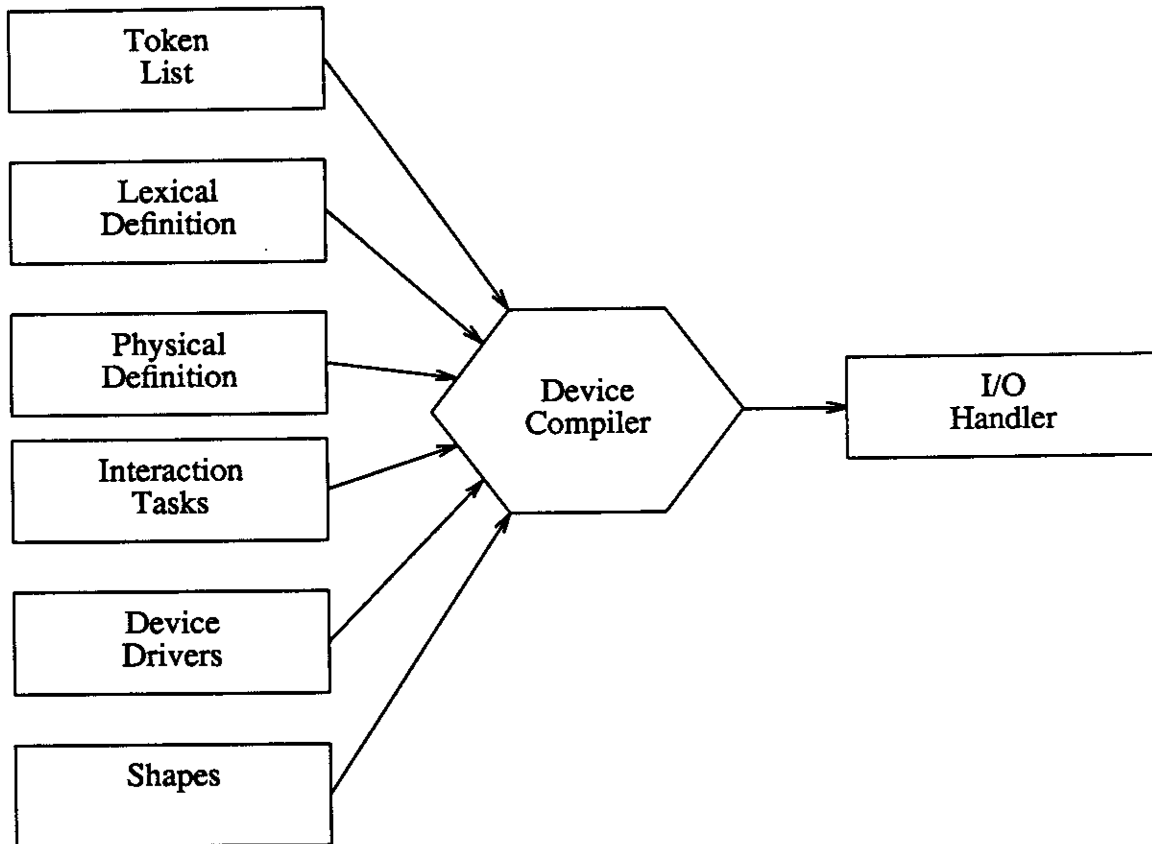


Figure 5.5: Device Compiler

5.9.2 Why LL(1)?

Although Fenchel had great success with SLR(1), LL(1) was chosen early in the research. An early goal of the prototype system was to support the wide variety of graphical interactions devices available at UCLA. It was found that some devices that supported both keyboard and mouse input could only enable one device at a time. Since we wish to allow a syntax designer to propose any sequence of device use, we need to know which device to enable, and we need to prompt for input if necessary. We later found that device independence, an honorable goal, was too large a task for our group. We continually found ourselves reducing sophisticated workstations to terminal emulators in our struggle for device independence.

The idea of *syntactic prescience* still permeated our work even after abandoning complete device independence as a goal. We always need to know where we are in the parse and where we are going. The need to unambiguously identify the end of a sentence is handled easily with existing LL(1) parsing techniques. The increased expressive power of the LR() family is desireable but had a significant drawback. During LR() parsing, it is possible to not know which of several rules will eventually be recognized. In fact, it may not be possible to detect which rule will be recognized until the first token of the next rule is input. This is no restriction at all to compilers of conventional programming languages since they typically needn't respond to the end user until the entire input program has been examined. It presents a severe problem if a variety of devices provide input, if a great deal of prompting is desired, or if an interactive user is to be told immediately that the input is in error.

Still, LL(1) is restrictive. The underlying ATN structure is capable of supporting any context free grammar and it may be desireable to relax the LL(1) restriction a bit.

A strong advantage of LL(1) languages is their *incremental compilation property*. A useful service of a command-response language interpreter is the ability for the end user to define new commands that are, perhaps, just slight variations on existing commands or even a single command that represents an aggregation of several system commands. Such user defined commands should be supported as any system defined command, that is, integral help, prompting, menu generation, etc., should be provided on all commands, not on just those defined by the system designer.

A nice property of LL(1) is that if a large syntax specification is LL(1) and if a small syntax specification fragment is LL(1), then the two can be merged easily and still be LL(1). Adding a new command to an LR(1) dialogue requires that the old

syntax and the syntax of the new command be merged and the entire syntax be recompiled.

5.9.3 Type Checking

The full implementation language is allowed within the action braces. The untyped nature of T makes type checking difficult. It is possible to write a separate processor to support type checking for lisp-like languages, but the idea is unpleasant to most LISP programmers.

The easiest solution is to pass the generated program, composed of action routines and parser, on to the compiler of a typed implementation language.

5.9.4 Prompting and Menus

A primitive, yet very useful, menu generation facility is provided in the prototype implementation. A more sophisticated implementation would include a menu description language that would allow the interface designer to specify any sort of menu presentation. A default menu object, such as those found in state-of-the-art workstations [Sun 84], could easily be provided for those interface designers that do not wish to involve themselves at that level.

5.9.5 Imperative Sentences and Vocative Expressions

Most commands to computer systems can be categorized as imperative sentences in the English language. That is, a directive is being issued to the subject of the sentence and the subject is assumed to be the computer system. The sentence "Do this." is equivalent to "You, do this." Another variation on the imperative sentence begins with a vocative expression, for example, "Ralph, do this." This variant is used when the imperative sentence is addressed to someone who may not otherwise

know that he or she is the recipient of the directive.

Recall that commands are part of a dialogue or a sub-dialogue and that each dialogue forms a encapsulation of token names defined within. Experience with the SARA-IDEAS system has shown that there is frequent need to issue a command to a tool other than the one currently being used. The Graph Model of Behavior tool is composed of several complementary tools and it is a common occurrence to wish to make a change to a model while using one of those tools and needing to reflect that change in another tool.

A useful construct to support in the command language is the vocative expression, such that the name of the external dialogue, followed by a comma, precedes the command that is desired. It should be clear by this discussion that the current dialogue is the assumed subject of all imperative sentences unless the imperative sentence is preceded by a vocative expression. The same treatment is appropriate for interrogative sentences.

5.9.6 Simultaneous Input Sources

The SARA-UIMS supports input from a variety of devices but only one device at a time. Very powerful graphics workstations currently on the market allow a user to, for example, manipulate two potentiometers simultaneously in order to perform a rotation of a display around two axes. A useful extension to this research is the treatment of multiple, simultaneous input streams.

5.9.7 Multi-party Dialogues

Another useful extension is the treatment of multi-party dialogues. Multi-party dialogues would allow a team of designers to interact with the design of a

model.

CHAPTER 6

The Logical and Physical Device Definition Phases

The device definition phases focus the tool developer's attention on the low-level aspects of the design tool's user interface. Neither of the previous phases, conceptual definition and syntax definition, specify the characteristics of any specific device, and are thus totally device independent. Device definition is composed of logical device definition and physical device definition, and deals exclusively with interaction techniques and devices. The device definition phases conclude the four-phase SARA-TBS method.

Each primitive notion (e.g., objects, terminal symbols) identified in the previous definition phases must come from some physical input device or must be displayed on some physical output device. A logical device definition buffers the developer from the specifics of physical devices with the more general interface characteristics of logical devices. Finally, a physical device definition bridges the gap between tool concept, syntax, logical interaction, and the physical devices with which the user comes in direct contact.

The chapter begins with an introductory discussion of terminology and issues involved at this level. Logical device definition and physical device definition are discussed separately. The device description of the influence diagram editor is given, followed by a syntactic description of the device definition language. The chapter concludes with a summary and suggestions for further work.

6.1 Introduction

The purpose of this section is to set the stage for the ensuing device description discussion. Relevant aspects from the conceptual definition phase and syntax definition phase are recalled and placed in the context of device definition. A first definition of interaction tasks, logical devices, and physical devices is presented and subsequently refined in later sections.

6.1.1 Interaction Tasks

An end user who is acclimated to input by alphanumeric keyboard may consider the fundamental input task to be depressing keys, or perhaps entering commands. An end user that has access to a system rich in input/output devices sees a variety of input tasks.

Input tasks at a higher level than the keystroke level are collectively known as *interaction tasks*. Foley [Fole82a] has proposed five basic interaction tasks: *locating* a position or viewpoint on a screen; *picking* a displayed entity from a screen; *valuating*, providing a single value from the space of real numbers; *keying*, providing character string input; and *selecting* one element of a small set, as from a menu.

6.1.2 Logical Devices

Each interaction task is supported by a *logical device*. The interaction task to logical device mapping is: locating - locator, picking - pick, valuating - valuator, keying - keyboard, and selecting - selector.

Each logical device has a natural analogue in the physical domain. The logical keyboard is naturally manifest as the common alphanumeric keyboard but could be supported by a voice input system. The logical pick device has as its natural analogue

the mouse or the pen and tablet. Mouse and tablet serve equally well as a physical realization of the logical locator device. A potentiometer or a keyboard is an obvious realization of the valuator. The programmed function key is the obvious analogue of the selector although the mouse and tablet can easily support the selecting task assuming that menus are displayed.

6.1.3 Physical Devices

Physical devices come from a variety of manufacturers, come in a variety of shapes, colors, costs, and sizes; and perform a variety of functions. One can be sure that even if two devices perform the same function, they do it differently. Both claim to do it better. Product specifications and trial-and-error are the final arbitrator of what and how a product performs.

Care must be taken in discussing physical devices. Specifically, a distinction must be made between a physical device and pieces of hardware packaged together. A single hardware package may contain more than one physical device.

For example, three physical devices built into two separate physical containers comprise an AED 512 station. The display device is contained in a single physical container, but the keyboard and joystick are built into the same physical housing.

Another example is a typical Evans and Sutherland PS300 station. One large display tube is provided as part of the PS300. The display screen can operate in either text mode or in vector graphic mode alternately or simultaneously. This one physical container thus supports two physical display devices. A separate *glass tty* is generally situated to the side for periodic communication with a host computer. A single physical container holds a sophisticated alphanumeric keyboard and a *banner display* running across the top of the keyboard suitable for labeling the eight function keys below

or for displaying general text. A separate package has six potentiometers, each with a small display just above it suitable for labeling the potentiometer. The last physical package is a tablet and pen.

For this PS300 station, there are six physical packages, display devices (2), keyboards (2), tablet (1), and potentiometer box (1); and 27 physical devices, CRT display devices (3), keyboards (2), tablet (1), potentiometers (6), banner display (1), function buttons (8), and potentiometer label displays (6).

6.1.4 Libraries

A tool building system such as SARA-TBS, much like an operating system, may exist in a variety of hardware environments. An operating system typically runs on a set of like central processing units with a single instruction set. However, the operating system may be required to manage an arbitrary amount of memory, an arbitrary number of mass storage devices, and a wide variety of terminals and printers. Operating systems manage this uncertainty by providing a large and diverse library of device drivers and an operating system design that decouples the primary operating system mechanisms from the organization and size of memory and input/output devices. SARA-TBS does the same.

Three libraries are part of the SARA-TBS library subsystem, the Device Dependent Library, DDL, the Device Independent Shape Library, DISL, and the Logical Device Library, LDL.

During tool execution, output takes place in three steps. The first step performed by the interaction handler is a call on a routine to draw an object, like a concept or an influence diagram. The graphics information is extracted from the object and passed as parameters to a routine that draws an appropriate shape, like a circle or

rectangle. Finally, the shape drawing routine makes the necessary calls on routines to draw generalized graphics primitives, like lines and polygons. Generalized graphics primitives are device dependent routines and are stored in the DDL. Shapes are dependent on a standard interface to generalized graphic primitives, but are device independent. Shape drawing routines are stored in the DISL.

Input is made through logical device drivers which are software routines that implement a logical device on a particular physical device. Logical device drivers are stored in the LDL.

6.1.5 Why the Introduction of Logical Devices?

Logical devices are software implementations of interaction tasks for a specific physical device. These logical input devices and logical output screens provide an important degree of freedom. A user who needs a high degree of prompting may wish to input tokens from the pick device while a more experienced user may wish to input tokens from the text device. Thus, it should be possible to change the binding of token groups from the %terminals section to different logical devices at run-time.

When a new physical device enters the inventory, a new logical device may be developed for it and stored in a library. This allows a tool system to accommodate a new physical device with a minimum of disruption.

An example of how this degree of freedom might be exploited is the adaptation of an interface to its user. The screen layout and physical device layout shown in Figure 6.1 is clearly biased towards a right handed user. The menu display is shown adjacent to the mouse. If a left handed user wished to move the mouse to the left side of the display, looking at the mouse and looking at the menu choices requires eye

movement from one extreme to the other. It may be quite costly to build a display configuration for both left and right handers, but it is quite easily handled in SARA-TBS. Tools only display to logical screens (as discussed in section 6.2). Affecting a change to the logical screen to physical device mapping software (as discussed in section 6.3) is all that is required in order to accommodate a left handed user. The tool's semantic routines need not be concerned with such details.

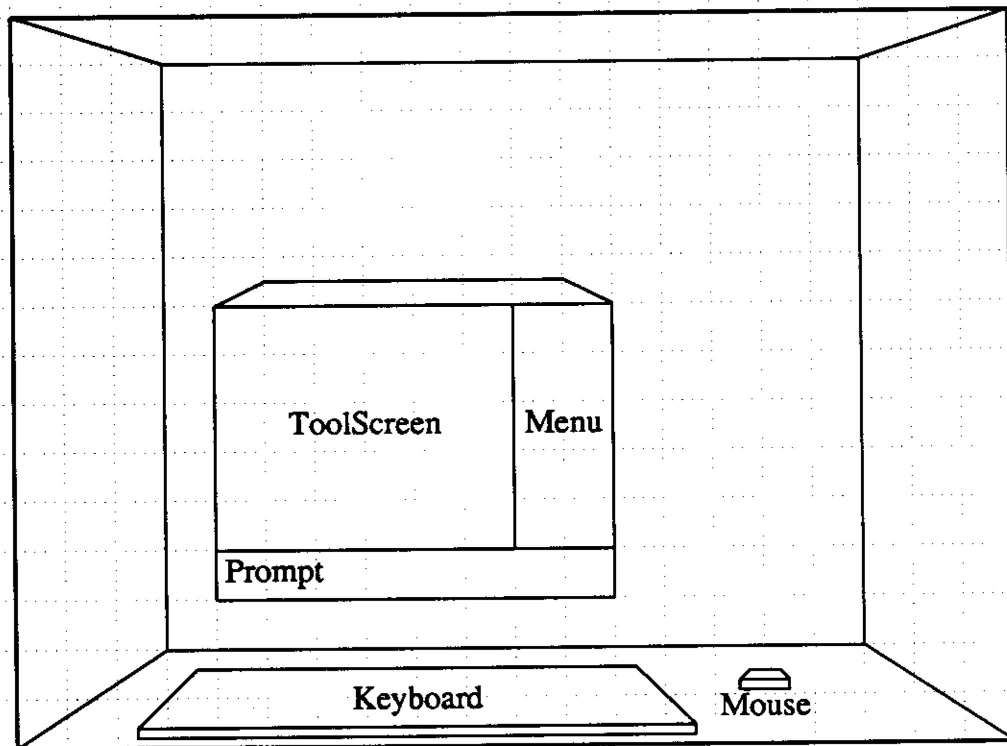


Figure 6.1: An Environment Alternative

6.1.6 Run-time Scenario

The Interaction Handler, IH, presented in chapter 7, is guided by the four phase description. It recognizes where it is in the dialogue by virtue of the ATN produced by the syntax definition phase. In order to continue the dialogue, the IH requests the next token as specified by the ATN. Each token belongs to a group of tokens as specified in the %terminals section of syntax specification. Each of those groups is associated with a logical device, for example, a pick device. Each logical device is, in turn, associated with a physical device, for example, the logical pick device is associated with a physical mouse.

Similarly, each output token has a graphical representation, a shape or icon, and will be displayed on some logical output screen, which, in turn, is implemented on some physical display device.

6.2 Logical Device Definition Phase

The previous section provides a discussion of interaction tasks and of logical and physical devices. This section concentrates on the specification of a tool's logical devices. Logical device definition buffers the developer from the specifics of physical devices with the more general interface characteristics of *logical devices* as discussed by Foley in [Fole82b].

First, the relevant portions of the previous syntax definition phase are recalled. The language is given for associating terminal symbols and icons, from the syntax phase, to logical input devices and to logical output devices, respectively.

6.2.1 Interaction Tasks

Input interaction tasks supported in the logical definition phase include locating, picking, valuating, keying, and selecting. Output interaction tasks supported in the logical definition phase include displaying and printing. Print can be viewed as a special case of display.

```
interactionTasks : inputInteractionTasks
   | outputInteractionTasks

inputInteractionTasks :
     locating
   | picking
   | valuating
   | keying
   | selecting

outputInteractionTasks :
     printing
   | displaying
```

6.2.2 Logical Input

First, let us recall the relevant syntax specification fragment from the influence diagram editor tool.

```
%terminals

verbs :
   structuredMode unstructuredMode
   addConcept    delConcept
   addInfluence  delInfluence
   quit ;

points :
   centerPoint ;

entities :
   concept influence influenceDiagram ;

strings :
   conceptName influenceProportion influenceDiagramName ;
```

The grammar compiler insured that all input terminal symbols were identified as either verbs, points, entities, or strings. Logical definition binds these groupings to a logical input device.

The logical definition begins with %logicalInputDevices and is composed of a series of simple associations, or rules. Each association has a left hand side, a colon, a right hand side, and a terminating semicolon. The left hand side is the name of one of the predefined logical input devices, pick, locator, valuator, keyboard, or selector.

```
logicalInputDevice :
    pick
  | locator
  | valuator
  | keyboard
  | selector
```

The right hand side is a sequence of one or more of the groupings from the %terminals section of the syntax definition. The logical input associations for the influence diagram editor are quite simple.

```
%logicalInputDevices
   pick : entities ;
   selector : verbs ;
   locator : points ;
   keyboard : strings ;
```

6.2.3 Logical Output

Next, let us recall the relevant specification fragment from the syntax specification of the influence diagram editor tool.

```
%rules
   ...
   cPick : concept ;
   iPick : influence ;
   cName : name ;

%icons
```

```
rectangle : influenceDiagram ;
circle : concept ;
polyline : influence ;
text : conceptName influenceProportion influenceDiagramName ;
```

The syntax definition phase introduces a list of objects that are part of the output syntax and provides a list of icons that are associated with those output tokens. The output tokens are influenceDiagram, concept, and influence. All output tokens were identified as either rectangles, circles or arcs. Logical definition binds the output tokens to a logical output device.

The logical output device definition for the influence diagram editor is that used for the SARA/IDEAS prototype.

```
%logicalOutputDevices
  toolScreen : influenceDiagram concept influence ;
  menuScreen : menu ;
  promptScreen : text ;
```

The logical definition begins with %logicalOutputDevices and is composed of a series of simple associations, or rules. Each association has a left hand side, a colon, a right hand side, and a terminating semicolon. The left hand side is the name of a logical output screen to be used by the tool. The right hand side is a sequence of one or more of the output tokens that might be displayed on this screen. The output tokens are taken from the %icons section of the syntax definition or are one of the primitive types supported by SARA-TBS.

6.2.4 Logical/Physical Interface

Discussion of the logical device definition phase concludes with a list of phase accomplishments and what remains to be done by the physical device definition phase.

All input tokens have been associated with a logical input device. A logical input device is defined by three operations, a *connector*, a *disconnector*, and a *getter*. These operations are roughly analogous to the operating system calls, open, close, and read. The arguments to the three operations differ depending upon the logical device being supported. The implementation of the logical device depends upon which physical device supports it. That determination is left to the physical device definition phase.

All output tokens have been associated with a logical display device, or screen. The icons or shapes that will be drawn on each screen have been identified. A logical output device is defined by three operations, a *connector*, a *disconnector*, and a *putter*. These operations are roughly analogous to the operating system calls, open, close, and write (append). The arguments to the three operations differ depending upon the logical device being supported. The implementation of the logical device depends upon which physical device supports it. The putter operation needs to know how to display all of the shapes that could be drawn on the associated logical screen. It is possible to define such drawing operations and store them in a device independent shape library if an interface to a device dependent set of drawing routines is defined in advance. At this stage of definition it is not known which physical device's set of device dependent drawing routines should be selected. That determination is left to the physical device definition phase.

6.3 Physical Device Definition Phase

Finally, the physical device definition phase focuses the developer's attention on the physical device inventory available to support interaction. Logical devices are assigned to physical devices. The ability to compose one or more physical displays into a single logical display, or dividing one physical display into several logical

displays is also defined in this phase.

6.3.1 Physical Input

The physical input specification associates logical input devices to the physical input device with which the end user will ultimately interact. Three associations are sufficient for the influence diagram editor.

```
pick : apolloMouse;
selector : apolloMouse;
locator : apolloMouse;
keyboard : apolloKeyboard;
```

Each association ends with a semicolon and has a left hand and a right hand side separated by a colon. The left hand side must be a name of a logical input device specified in the logical input device section. The right hand side is a name sufficiently unique to select a library entry from the LDL.

6.3.2 Physical Output

Physical output is a bit more complex to describe than physical input. Two separate phenomena must be dealt with; the partitioning or aggregating of physical devices into displays (the displays section), and mapping logical screen names, introduced in the logical output device section, onto displays (the screens section).

6.3.2.1 Display Surfaces

The displays section begins with %displays and contains rules. The rules are more complex than in previous sections because they must be able to express a one-to-one, one-to-many, or many-to-one correspondence between physical screens and display surfaces. Typically, the left hand side introduces new names. Following a colon, the right hand side contains a name of a physical display device. The device

name is used to search a library of device descriptions. Library entries include device height and width, device address granularity, device address translation functions, and other characteristics.

The following simple fragment is from the SARA/IDEAS prototype implementation. It introduces the new name "displaySurface" and specifies a one-to-one correspondence between that name and the physical output device description of "apolloScreen".

```
%displays
   displaySurface : apolloScreen ;
```

In general, a single display may be supported by a single physical display device, by a part of a single display device or by a collection of physical display devices.

Rather than divide a single physical display device into several display surfaces, we sometimes wish to simulate a large screen display by composing many physical display devices into one large display surface. The following %displays specification aggregates six physical display devices into one display surface.

```
%displays
   displaySurface[ 0/3 : 1/3; 0/2 : 1/2 ] : sony /* s1 */
   displaySurface[ 1/3 : 2/3; 0/2 : 1/2 ] : sony /* s2 */
   displaySurface[ 2/3 : 3/3; 0/2 : 1/2 ] : sony /* s3 */
   displaySurface[ 0/3 : 1/3; 1/2 : 2/2 ] : sony /* s4 */
   displaySurface[ 1/3 : 2/3; 1/2 : 2/2 ] : sony /* s5 */
   displaySurface[ 2/3 : 3/3; 1/2 : 2/2 ] : sony /* s6 */
```

The end result of the display section is a list of display surfaces which is carried forward to the next section.

6.3.2.2 Logical Screens on Display Surfaces

The screens section begins with %screens and contains rules. Each rule has a left hand side, a colon, a right hand side, and a terminating semicolon. Each rule defines what portion of a display is occupied by a logical screen. The left hand side is the name of a logical tool screen introduced in the logical output definition. The right hand side begins with a name introduced in the %displays section and is followed by a two part expression. The expression is contained between brackets ([]) and the first and second parts are separated by a semicolon. The first part of the expression describes what portion, along the horizontal X axis, of the display surface is allocated to the logical screen. The second part of the expression similarly describes the assignment along the Y axis. Each expression has two parts separated by a colon. The first part of an expression describes the starting point along the appropriate axis, and the second part describes the ending point. Either floating point or common fractions are allowed.

For example, in the SARA-IDEAS system, the screen of an APOLLO workstation is divided into three different logical screens. In our specification language, this is expressed as

```
%screens
  ToolScreen   : displaySurface[ 0.0 : 0.8; 0.0 : 0.9 ] ;
  MenuScreen   : displaySurface[ 0.8 : 1.0; 0.0 : 1.0 ] ;
  PromptScreen : displaySurface[ 0.0 : 0.8; 0.9 : 1.0 ] ;
```

The physical screen is represented by coordinates from 0.0 to 1.0 and every logical screen occupies a fraction of the physical screen by assigning it a fraction in the interval 0.0 to 1.0.

Combining the first %displays section with this %screens section yields

```
%displays
   displaySurface : apolloScreen ;

%screens
   ToolScreen   : displaySurface[ 0.0 : 0.8; 0.0 : 0.9 ] ;
   MenuScreen   : displaySurface[ 0.8 : 1.0; 0.0 : 1.0 ] ;
   PromptScreen : displaySurface[ 0.0 : 0.8; 0.9 : 1.0 ] ;
```

and produces an APOLLO screen with the layout shown in Figure 6.2.

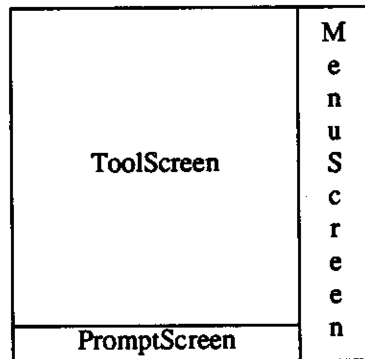


Figure 6.2: Sample Screen Layout

Combining the second %displays section with this %screens section yields

```
%displays
   sony1 sony2 sony3 sony4 sony5 sony6 : sony

   displaySurface[ 0/3 : 1/3; 0/2 : 1/2 ] : sony1
   displaySurface[ 1/3 : 2/3; 0/2 : 1/2 ] : sony2
   displaySurface[ 2/3 : 3/3; 0/2 : 1/2 ] : sony3
   displaySurface[ 0/3 : 1/3; 1/2 : 2/2 ] : sony4
   displaySurface[ 1/3 : 2/3; 1/2 : 2/2 ] : sony5
   displaySurface[ 2/3 : 3/3; 1/2 : 2/2 ] : sony6

%screens
   ToolScreen   : displaySurface[ 0.0 : 0.8; 0.0 : 0.9 ] ;
   MenuScreen   : displaySurface[ 0.8 : 1.0; 0.0 : 1.0 ] ;
   PromptScreen : displaySurface[ 0.0 : 0.8; 0.9 : 1.0 ] ;
```

and produces a screen like the one shown in Figure 6.3.

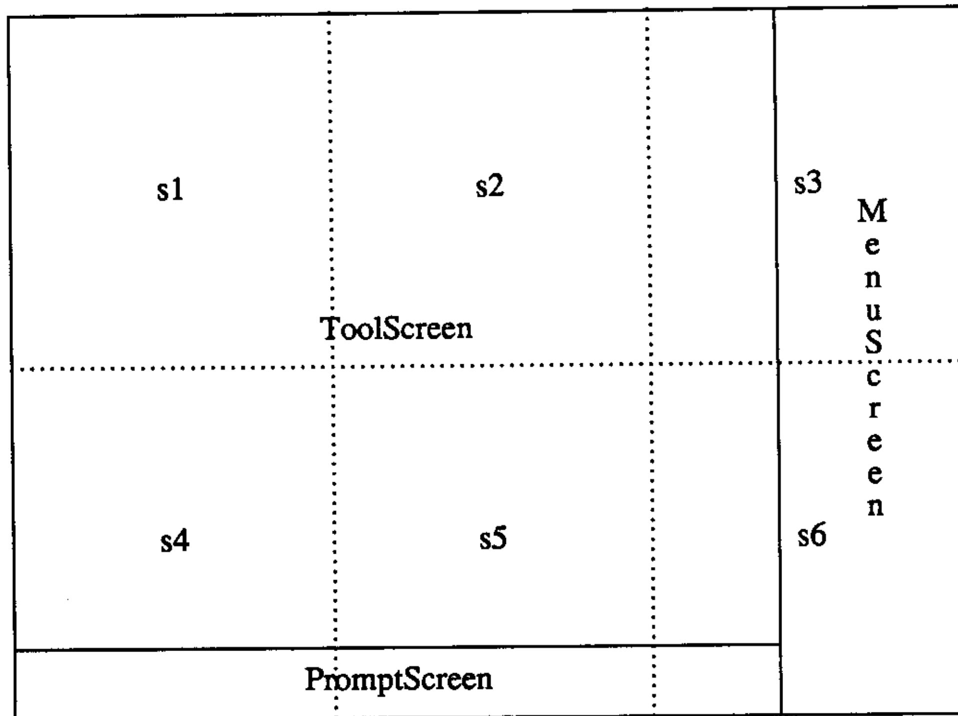


Figure 6.3: Many Physical Devices Comprising One Display Surface

6.4 Interaction Environment

Several other aspects of the physical environment may also be dealt with in this phase. For example, the spatial arrangement of physical devices relative to the user may be specified. Given the syntax of the command language, mapping of grammar terminal symbols to logical devices, mapping of logical devices to physical devices, and the spatial arrangement of physical devices, we can statically or dynamically measure the hand and eye motions required to conduct a dialogue. These measurements may lead to a better understanding of what constitutes a good interface. It is not the intention here to describe what constitutes a good user interface, but it does seem appropriate to provide a means by which user-interface experiments can be conducted.

The facilities described so far allow a tool developer to provide many different interfaces by specifying different grammars and different combinations of logical and physical devices. Given spatial arrangement and physical dimensions of physical devices, it is possible to allow a tool developer to propose several interfaces and easily gather experimental data on the use of the interfaces.

As mentioned previously, the library entries that describe a physical device contain a variety of physical device characteristics including physical dimensions. If the tool developer extends the physical device definition to include the geometry of the interaction environment, useful interface measurements can be extracted.

What measurements might be taken? The amount of end user hand and eye movement are obvious candidates, and are called *tactile alternation,* and *visual alternation* respectively [Bles82]. *Keystroke level errors [Card83]* have been proposed as a figure of merit for user-interfaces as well as *appropriate gesture [Feld82],* the appropriateness of the human's gesture for the interaction task. Each measurement is discussed below.

6.4.1 Tactile Alternation

Input requires some physical action (including retinal monitoring, voice recognition, and other exotic inputs) from the user. If we take the sentence to be the basic unit of dialogue, then we might expect that a sentence that requires very little physical movement from the user is preferred to a sentence that requires a great deal of movement. Each sentence is a non-terminal but eventually expands to a finite sequence of terminal symbols, each of which comes from a logical input device and subsequently from a physical input device. The cost in hand movement can be computed from the number of alternations from one logical or physical device to another. If each physi-

cal device description in the library subsystem contains physical measurements, and the relative positions of physical devices comprising the interaction environment are known, it is possible to measure an alternation from one device to another in physical distances.

Section 6.4.5 provides a simple mechanism to support the measurement of tactile alternations. Given a map describing relative positions of devices we can extract the physical hand movements a user must perform in order to navigate through a sample session. This measurement can be determined statically without executing the system, in fact, without the system even being built. Only the specification need be provided. A sample session can be a monitored session of an end user working at a real problem on the final system or it can be a set of sentences selected from the grammer. Sentences from the grammar can be selected at random, according to their anticipated frequency of use, or according to actual frequency of use as reflected by statistics gathered from a previous system.

Implementation and integration of the semantic operations can proceed in parallel, or be postponed, while the interface is being evaluated.

6.4.2 Visual Alternation

Visual alternations are the result of the end user's eye movement over the device terrain. It is similar to tactile alternation, but measures the end user's eye movement over display devices rather than over input devices. An interface that requires a great deal of eye movement between the menu screen and the tool screen may cause eye strain or it may cause the end user to remain alert [Card83]. In either case, being able to measure the visual alternations of an interface allows different interfaces to be compared and evaluated.

Section 6.4.5 describes a mechanism for supporting measurement of visual alternations.

6.4.3 Keystroke Errors

The number of errors committed at the keystroke level [Card83] has been proposed as a metric of user-interface quality. The research leading to the keystroke model was conducted when the keyboard was the primary input device, but the underlying model is still applicable. The basic idea is that user errors are an indication of a difficult or poor user-interface. The SARA-UIMS system operations, discard token, dt(); and discard sentence, ds(); are easily augmented to capture this information.

6.4.4 Appropriate Gesture

Appropriate gesture [Feld82] is a user-interface measurement more recently proposed. To make the notion clear, consider the situation where a human rearranges items on a table and the analogous situation where a human rearranges icons on a screen using a mouse.

If a human wishes to move an item from one place on a table to another, the human gesture will be to grasp the item and hold on to it while moving the hand and object to the desired place and then letting go of the item. If a human wishes to move an icon from one place on a screen to another, one would expect the sequence of events to be similar. However, the typical sequence is to move the hand holding the mouse until the cursor is over the desired icon (analogous so far), push a button (grasp) and then let go of the button (inappropriate gesture), move the hand and mouse (move) to the desired location, then push the button again to indicate letting go of the icon (another inappropriate gesture). A more appropriate gesture is to click the mouse (grasp) and keep the button depressed (hold) until the cursor is positioned

(move), and then let go of the button (release) to indicate letting go of the icon.

SARA-TBS offers no support for appropriate gesture measurement even though it is an interesting concept.

6.4.5 Physical Interaction Environment Specification

The SARA-TBS prototype does not include the following extension to the specification language. The current system supports only two-dimensional graphics, but nothing in the implementation precludes the addition of a third dimension. With three dimensional graphics, the following language would be replaced with a graphics tool.

The language of PIC [Kern77] is the basis for what follows.

The SARA-IDEAS interaction environment, represented in Figure 6.4, is described below.

```
%environment

env is 46 wide, 40 high, 46 deep.
apolloScreen.f.l.c at (22.5,8,16);
apolloKeyboard.d.b.c at apolloScreen.d.f.c+(0,0,-5);
apolloMouse.d.l.c at apolloKeyboard.d.r.c+(6,0,0);

eye at apolloScreen.f.c+(0,6,-13);
lefthand at apolloKeyboard.f.c+(-2.5,.5,0);
righthand at lefthand+(5,0,0)
```

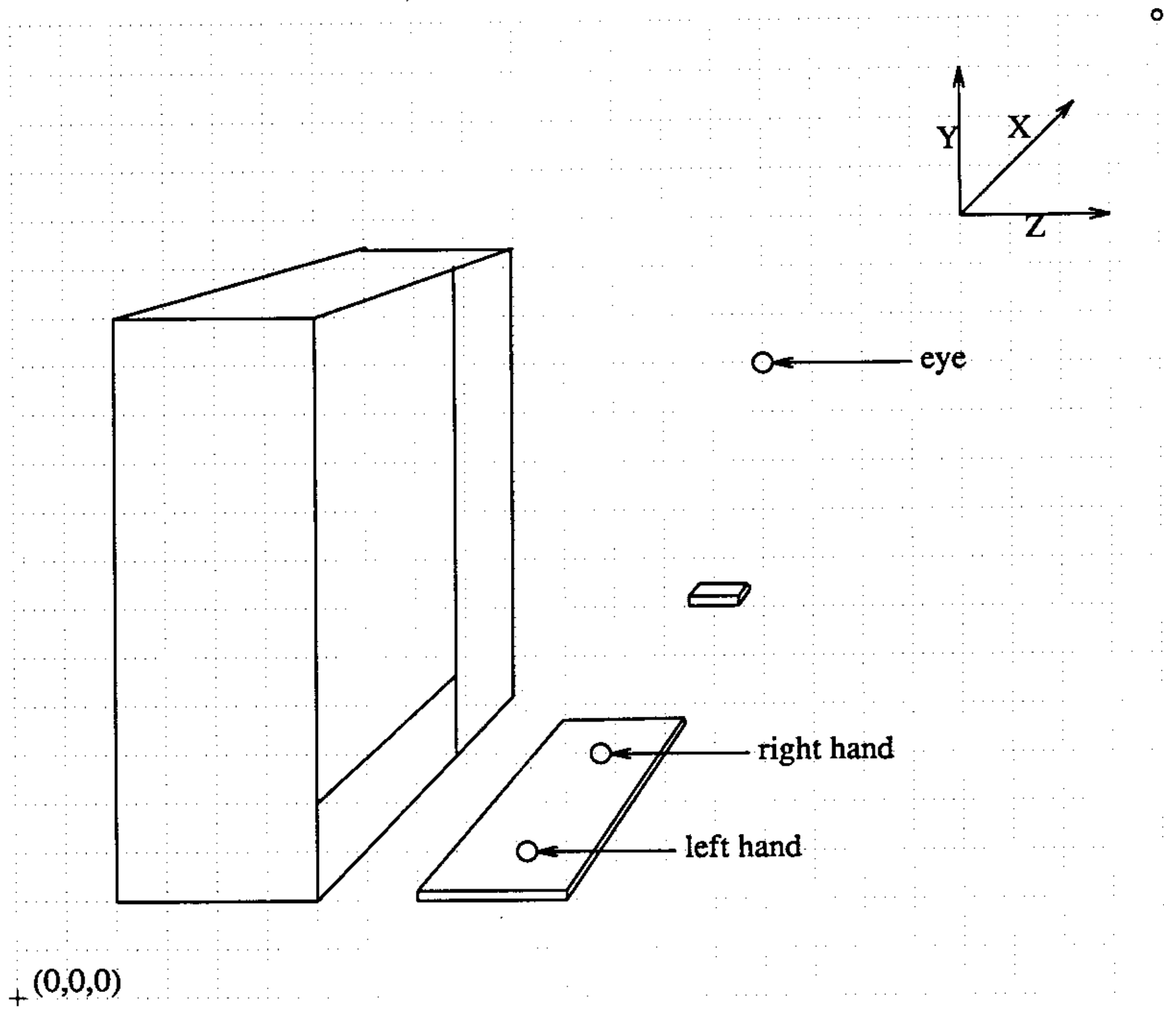


Figure 6.4: Interaction Environment

The following YACC syntax specification provides a possible, primitive language for interaction environment definition. A separate lexical analyzer is responsible for recognizing floating point numbers (FLOAT) and names (NAME).

```
%%
environmentDefiniton : '%environment'
   OptionalConstantDefinitions
   SpatialArrangement

OptionalConstantDefinitions : /* empty */
   | ConstantDefinitions
ConstantDefinitions : ConstantDefinition
   | ConstantDefinitions ConstantDefinition
ConstantDefinition : ConstantName '=' Value ';'
```

```
SpatialArrangement : EnvironmentDefinition DevicePlacement UserPlacement

EnvironmentDefinition :
   'env' 'is' Width ',' Height ',' Depth OptionalVanishingPoint '.'
OptionalVanishingPoint : /* empty */
   | 'with' 'vanishing' 'point' Placement

DevicePlacement : OptionallySuffixedDeviceName Placement ';'

UserPlacement : EyePlacement HandPlacements
HandPlacements :
     LeftHandPlacement RightHandPlacement
   | RightHandPlacement LeftHandPlacement

EyePlacement : 'eye' Placement
LeftHandPlacement : 'left' 'hand' Placement
RightHandPlacement : 'right' 'hand' Placement

OptionallySuffixedDeviceName :
     DeviceName
   | DeviceName Suffixes

Placement : 'at' Point3d

Suffixes : Suffix
   | Suffixes Suffix
Suffix :
     '.l' /* left   */
   | '.r' /* right  */
   | '.u' /* up     */
   | '.d' /* down   */
   | '.f' /* front  */
   | '.b' /* back   */
   | '.c' /* center */

Point3d : '(' x ',' y ',' z ')'
   | OptionallySuffixedDeviceName
   | Point3d PlusOrMinus Point3d

Width  : x 'wide'
Height : y 'high'
Depth  : z 'deep'

x : coordinate
y : coordinate
z : coordinate

coordinate : FloatExpression

Value : FloatExpression
FloatExpression : Float
   | '(' FloatExpression ')'
```

```
  | FloatExpression Operator FloatExpression
Float : NUMBER

DeviceName : NAME
ConstantName : NAME

Operator : '+' | '-' | '*' | '/'
PlusOrMinus : '+' | '-'
%%
```

6.5 Device Libraries

Three libraries are specified to meet the requirements of device independence, reusability, and avoidance of a lowest-common-denominator approach. The three libraries comprise the set of input/output libraries. They are: the device dependent library, DDL, where software packages supporting low level graphic display are stored; the device-independent shape library, DISL, where shape drawing routines are stored; and the logical device library, LDL, where software packages supporting logical device interaction are stored.

The system librarian is represented in Figure 6.5 and accepts two kinds of commands, search commands and install commands. As new input/output routines are developed, they can be installed in the library. The device definition phase compiler makes search calls on the librarian to withdraw a specific routine or collection of routines as specification analysis determines the need.

6.5.1 Device Dependent Library (DDL)

This library contains packages of functions and data that are sufficient to draw generalized graphics primitives, GGPs, on a specific physical display device. DDL entries are thus tool independent and device dependent. It was previously planned to search this library for individual functions on an as-needed basis. The functions

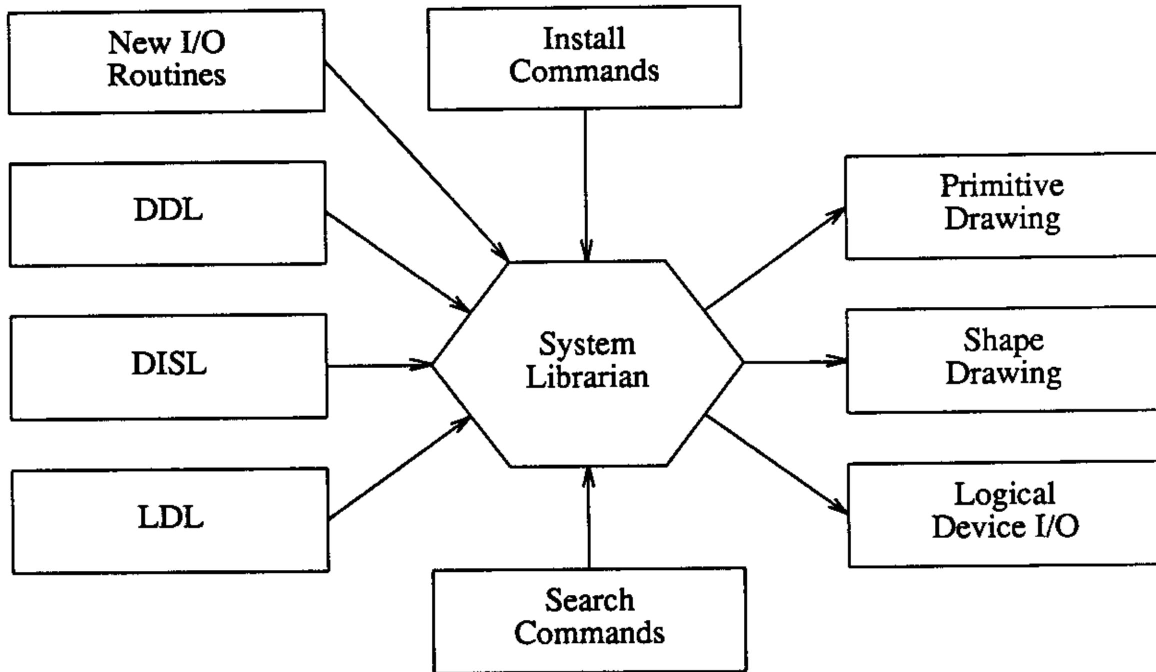


Figure 6.5: System Librarian

would have to be side-effect-free if they were to be treated separately. The current organization draws a cooperating collection of functions and data from the DDL. As an example of why pure functions were inadequate, consider the case of the current-cursor-position. Almost all drawing functions alter the cursor position. This variable needs to be defined, initialized, and shared. A partial list of functions follow.

```
(drawPolygon listOfPoints)
(drawPolyline listOfPoints)
(drawCircle centerPoint radius)
(drawSpline listOfPoints)
(drawText text position font)
```

These functions should have no knowledge of higher level object data structuring, e.g., all arguments should be extracted from the object CONCEPT before calling drawCircle.

The T call to search the DDL is:

```
(search-ddl physical-device)
```

and the entire function/data package is returned.

6.5.2 Device-Independent Shape Library (DISL)

The Device-Independent Shape Library, DISL, contains routines that draw *shapes*. An example of a shape is a rectangle as opposed to the higher level object *influence diagram* or the lower level generalized graphic primitive, GGP, *polygon*. This library will accumulate over time and its contents are only required to contain sufficient definitions for the current tool system. Subsequent tool development should be able to exploit library entries developed for previous tools.

These library entries are pure function and only those required by the current tool are withdrawn from the library. These functions are device independent in that they are defined in terms of GGPs bound together with the implementation programming language, T.

```
(search-disl shape device-capability-category)
```

Shape is one of {circle, rectangle, hexagon, ...}. The complete list is determined by the tool under development. The parameter device-capability-category is added to allow another level of flexibility.

The device-capability-category variable may take on values like *primitive, core, gks,* etc. This allows for more than one implementation of a shape. As an example, consider the capabilities of the GT101 or similar semi-dumb terminals. They can not draw a circle but have some graphics capabilities. The GT101 can draw horizontal and vertical lines and right angle corners. With these GGPs we can define

a nice rectangle, a shape. A more powerful display device would implement this much more effectively as a polygon.

The lowest-common-denominator, LCD, approach would suggest that we not implement GGPs as polygons but as horizontal lines, vertical lines, and corners. We wish to avoid that approach by aiming at a more sophisticated device. On the other hand we do not wish to preclude the use of devices that a user may have at home. The device-capability-category addresses that dilemma. Since CORE and GKS already provide the facilities needed, their selection is also allowed.

6.5.3 Logical Device Library (LDL)

This library, unlike the previous two, deals with both input and output. The library is searched by

```
(search-ldl logical-device physical-device . options)
```

A logical device is one of {locator, pick, valuator, keyboard, selector, display, printer}. The search returns a software implementation, the logical device, of the requested logical device type on the requested physical device.

The portion of the specification language dealing with physical device inventory and layout allows the binding of physical devices in the inventory to the logical device function they are to perform.

Each terminal symbol in the grammar is by definition an indivisible unit that the user must provide as input. Associated with each of these is a *getter* function. The interaction handler, IH, invokes the appropriate getter guided by the parse. The following functions comprise the set of getters.

1. select-category

2. select-verb

3. select-entity

4. get-point

5. get-value

6. get-string

Getters 1-3 may all be the same but certainly make a call on (xy->container (get-point locator)). Thus, pick is implemented in terms of the locator.

The specification language allows binding of terminal symbols to one of {category, verb, entity, point, value, string}. As the compiler encounters these bindings it can store the type with the terminal symbol. Therefore, each terminal symbol is an object of one of the above types and its getter is specified accordingly.

6.6 Influence Diagram Editor Device Description

This section contains the complete device definition for the influence diagram editor. First the %terminals and %icons sections from the syntax phase are given, followed by the logical device defintion and finally the physical device definition.

```
%terminals

verbs :
   structuredMode unstructuredMode
   addConcept   delConcept
   addInfluence  delInfluence
   quit ;

points :
   centerPoint ;

entities :
   concept influence influenceDiagram ;
```

```
strings :
   conceptName influenceProportion influenceDiagramName ;


%rules
   ...
   cPick : concept ;
   iPick : influence ;
   cName : name ;

%icons

   rectangle : influenceDiagram ;
   circle : concept ;
   polyline : influence ;
   text : conceptName influenceProportion influenceDiagramName ;


--- logical definitions
%logicalInputDevices
   pick : entities ;
   selector : verbs ;
   locator : points ;
   keyboard : strings ;

%logicalOutputDevices
   toolScreen : influenceDiagram concept influence ;
   menuScreen : menus ;
   promptScreen : strings ;

--- physical definitions
%inputDevices
   pick : apolloMouse ;
   selector : apolloMouse ;
   locator : apolloMouse ;
   keyboard : apolloKeyboard ;

%displays
   displaySurface : apolloScreen ;

%screens
   toolScreen   : displaySurface[ 0.0 : 0.8; 0.0 : 0.9 ] ;
   menuScreen   : displaySurface[ 0.8 : 1.0; 0.0 : 1.0 ] ;
   promptScreen : displaySurface[ 0.0 : 0.8; 0.9 : 1.0 ] ;
```

6.7 Complete Device Description Languages

This section contains the YACC syntax specification of the device languages. NAME is a terminal symbol that the lexical analyzer is expected to recognize after consulting the symbol table. The lexical analyzer should be able to categorize a NAME as being one previously identified as an icon, physical input device, physical output device, display, or logical screen. FRACTION is either a decimal fraction D, 0.0 <= D <= 1.0, or a common fraction, C, 0/1 <= C <= 1/1. FRACTIONs are also recognized by the lexical analyzer.

```
%%
DeviceDefinition : LogicalDeviceDefinition PhysicalDeviceDefinition
LogicalDeviceDefinition : LogicalInputDevices LogicalOutputDevices
PhysicalDeviceDefinition : PhysicalInputDevices PhysicalOutputDevices

LogicalInputDevices  :'%logicalInputDevices'  LIDrules
LogicalOutputDevices  :'%logicalOutputDevices'  LODrules
PhysicalInputDevices  :'%inputDevices'  PIDrules
PhysicalOutputDevices :DisplaysSection ScreensSection

DisplaysSection : '%displays' DisplayRules
ScreensSection : '%screens' ScreenRules

LIDrules : LIDrule | LIDrules LIDrule
LODrules : LODrule | LODrules LODrule
PIDrules : PIDrule | PIDrules PIDrule
DisplayRules : DisplayRule | DisplayRules DisplayRule
ScreenRules : ScreenRule | ScreenRules ScreenRule

LIDrule : LogicalInputDeviceName ':' InputTypes ';'
LODrule : LogicalScreenName ':' IconTypes ';'
PIDrule : LogicalInputDeviceName ':' PhysicalInputDeviceName ';'
DisplayRule : OneToOneDisplayRule
   | AggregatingDisplayRule
   | PartitioningDisplayRule

OneToOneDisplayRule : DisplayName ':' PhysicalOutputDeviceName ';'
AggregatingDisplayRule :
   DisplayName XYpartition ':' PhysicalOutputDeviceName ';'
PartitioningDisplayRule :
   DisplayName ':' PhysicalOutputDeviceName XYpartition ';'

ScreenRule : LogicalScreenName ':' XYpartition ';'

LogicalInputDeviceName :
```

```
    'pick'
  | 'locator'
  | 'valuator'
  | 'keyboard'
  | 'selector'

InputTypes : InputType | InputTypes InputType
InputType :
    'category'
  | 'operation'
  | 'entity'
  | 'position'
  | 'quantity'
  | 'text'

LogicalScreenName : NAME
DisplayName : NAME
PhysicalOutputDeviceName : NAME
PhysicalInputDeviceName : NAME
IconTypes : IconType | IconTypes IconType
IconType : NAME

XYpartition : '[' FractionPair ';' FractionPair ']'
FractionPair : FRACTION ':' FRACTION

%%
```

6.8 Device Compiler

The device compiler analyzes the logical device definition and the physical definition to produce an input/output handler for the tool being specified. In addition to the device definitions, the compiler consults the token list produced by the syntax phase. Rather than creating new software, the device compiler makes calls on the system librarian for predefined software packages. Figure 6.6 shows the data flow diagram of the device compiler.

6.9 Summary and Conclusion

The logical device and physical device definitions comprising the complete device definition for the influence diagram editor have been given. The specification

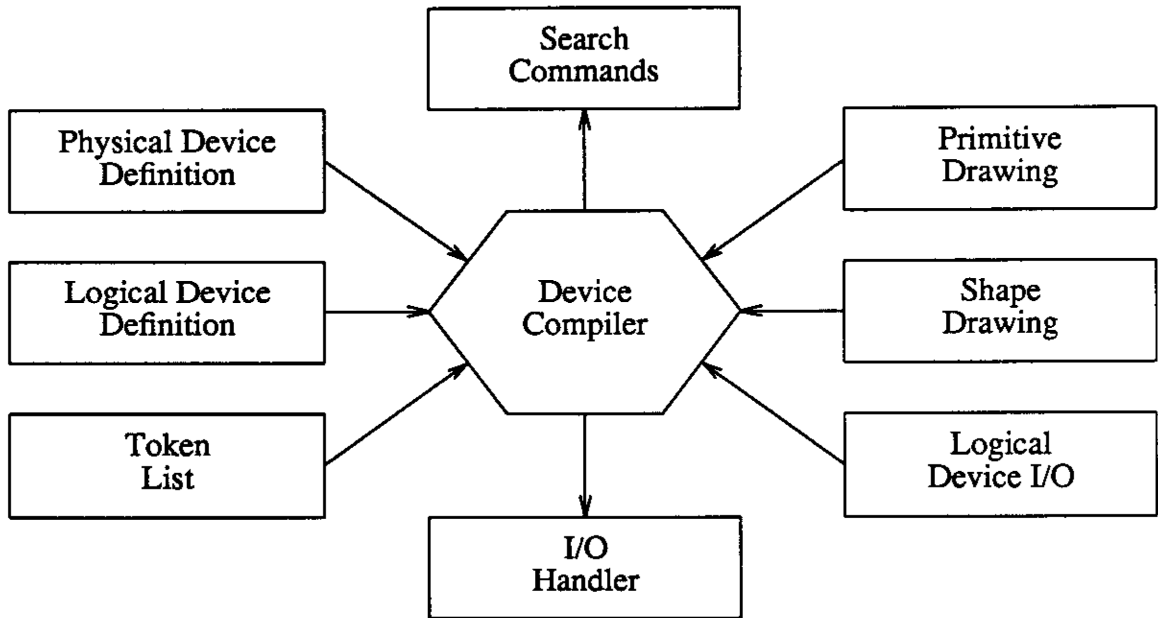


Figure 6.6: Device Compiler

fits on one page which is as it should be. The complexity of input/output software is hidden from the designer of a new tool and made the responsibility of a systems programmer. Input/output software is thus done in a general fashion rather than in a manner that makes it suitable for only one tool. As new devices enter the inventory, input/output software is developed and stored in the appropriate library where it becomes immediately available to any tool that has been developed using the SARA-TBS method.

The SARA-TBS fails to achieve device independence. The goal of device independence was replaced by *device tolerance*. Until the rapid rate of advancement being made in computer interaction devices slows, standardization of device interfaces will not occur and device independence will not be achieved.

The environment definition language should be implemented and user interface experiments begun. Only then will the requirements of a user interface experi-

mentation system become clear. What makes a good interface is currently being discussed philosophically. The pros and cons of user interface are rarely inferred from rigorous experimentation. Perhaps the reason for this is that it is prohibitively expensive to build special tools properly instrumented for experiment. SARA-TBS provides an environment for the development of real tools that could allow user interface experimentation at a negligible cost.

CHAPTER 7

The SARA User Interface Management System

Previous chapters present the SARA Tool Building System, SARA-TBS. A phased method is given for describing the aspects of one tool that distinguish it from other tools. Each phase of the method provides: a framework for discussing the semantics, the syntax, the logical device, and the physical device characteristics of a single tool, a specification language for describing each characteristic of that tool, and finally a translator that takes the specification into an implementation. Implicit in SARA-TBS is the existence of a reconfigurable target machine providing a powerful execution environment of highly integrated, sharable components. This chapter describes the reconfigurable target machine, the SARA User Interface Management System, SARA-UIMS.

The chapter begins by making a distinction between policy and mechanism. A list of UIMS supported mechanisms is given. Next, the user interface management system is decomposed into a command-response system, a state representation system, and a reference system. Each is described separately. Finally, conclusions are drawn and suggestions are given for further research.

7.1 Introduction

This dissertation describes a method for the economical construction of CADOCS tools and their integration into a system of CADOCS tools. Clearly, any two tools must be different in some, probably many, respects. Developers of a CADOCS tool system do not know *a priori* what new tools will be proposed for consideration in the future and therefore can not discuss what their differences will be. They can, however, be assured that any new interactive CADOCS tool, for example, will have a command language and will allow users to interact with the tool through physical devices. Of course any two tools will offer their users a different set of commands, and, over time, the set of physical devices will change.

The developers of a CADOCS tool building system must consider what is *fixed* regardless of the CADOCS tool and what is *variable* between CADOCS tools. The need for a command language is fixed. The specific command language syntax and semantics is variable. The need to interact through physical input and output devices is fixed, the set of physical devices supporting interaction is variable. The fixed portions identified in this chapter have policy independent mechanisms to support them. Policy decisions are left for the tool designer to consider along with the variable portion.

All CADOCS tools require certain mechanisms that, if built correctly, can be shared by a collection of CADOCS tools. By "built correctly" we mean that the mechanism is built independent of any particular policy and independent of any specific tool's application.

The remainder of this section identifies what functionality is desirable and permissible in the UIMS, and what functionality must be specified via the definition

languages presented in previous chapters. A distinction between *policy* and *mechanism* is made clear. The UIMS components are identified. Finally, the organization of the remainder of the chapter is presented.

7.1.1 Policy versus Mechanism

Interactive, time-sharing operating systems manage the work of many users. One responsibility of the operating system is to equitably divide host computer resources between users. In this context, there are many ways to define ''equitably''. Discussing alternative means of equitable resource allocation is used to clarify the distinction between policy and mechanism.

There are many familiar cpu scheduling algorithms employed in operating systems. *First-Come-First-Served,* FCFS, *Shortest-Job-First,* SJF, and *Round-Robin,* RR, are common examples [Pete83a]. If FCFS is employed, jobs are queued in the order they arrive. When the cpu becomes available, the oldest job in the queue is selected for execution and run until completion. If SJF is used, the shortest job is selected for execution and run until completion. If RR is used, each job is selected in turn, and, rather than run until completion, is run for a system-wide amount of time, the *time quantum.*

User jobs are usually referred to as processes internal to the operating system. Processes may be in one of several states, e.g., active, blocked, sleeping, or terminated, they belong to a user, they may communicate with other processes, and they may have open files. With each process is associated a Process Control Block, PCB. The PCB has separate components capable of representing each of the process' state, owner, open files, etc. The operating system contains a set of procedures and functions to read and write the value of PCB fields.

The set of PCB procedures and PCB functions and the PCB data structure itself comprise the mechanism associated with process management. The algorithms, FCFS, SJF, RR, represent the system's chosen policy towards process management. The mechanism remains fixed while the policy selected varies.

As a second example of a policy independent mechanism, let's hypothesize that the need for interactive user error recovery warrants a UIMS mechanism to regulate state change. The work space is the data structure that represents state. We may conclude that **do, redo,** and **undo** are operations that surround the data structure and are the only way of accessing the data structure. Together, the operations and data structure comprise a state representation mechanism as suggested in Figure 7.1.

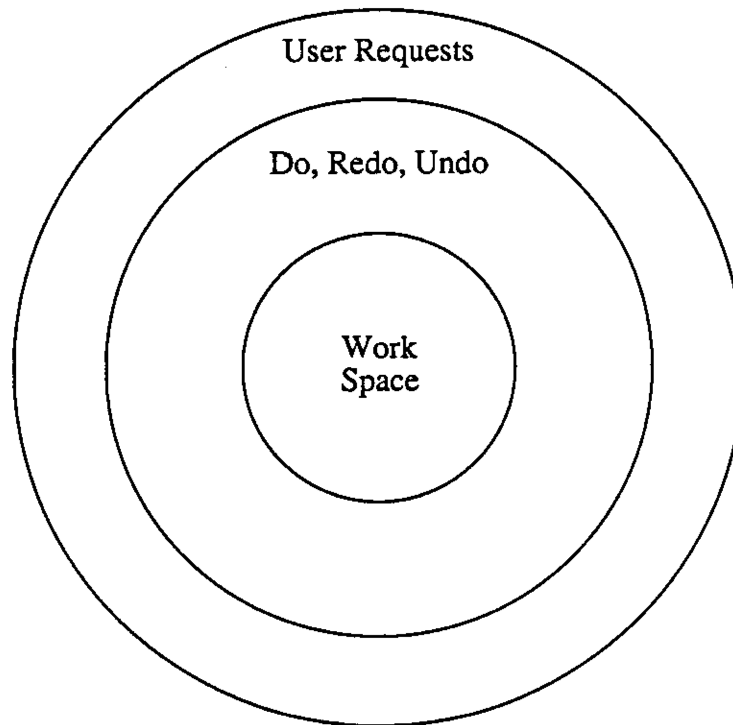


Figure 7.1: State Representation Mechanism

Two examples of a state representation mechanism should make the definition of a policy independent mechanism clear. Both examples are taken from the BSD 4.2 Unix.

The VIsual, VI, editor [Joy80a] allows its users to **do** a wide variety of commands that will query or alter its work space, in this case, the buffered copy of the file under edit. Examples of such commands are the global substitution of one text string for another, the deletion of one or more lines of text, and the insertion of one or more lines of text between two existing lines of text. VI also allows users to **undo** edit commands. VI's policy is to allow only the most recent work space altering command to be **undone**. Thus, all commands other than the most recent command are **commit**ted and cannot be **undone**. In this case, the mechanism is built to support only the policy presented to the user.

In the interactive command interpreter, the C-Shell, CSH [Joy80b], the work space is composed of shell variables and the file system. CSH allows the **do**ing and **redo**ing of commands. A script of previously issued user commands is maintained by CSH as a circular history queue, HQ, of size H. Users are free to select the value of H, but 16 is a representative value as shown in Figure 7.2.

When a command is issued the command script is copied into HQ[next] and next (the queue index) is advanced clockwise. Thus, the most recent H commands are available for **redo**ing. In this case, the system offers a fixed mechanism and allows the user to make the policy decision of how many commands are retained for **undo**ing by varying the value of H.

It is interesting to note that CSH supports **redo** of the last H commands and **undo** of the last zero commands while VI supports **undo** of the last one command and

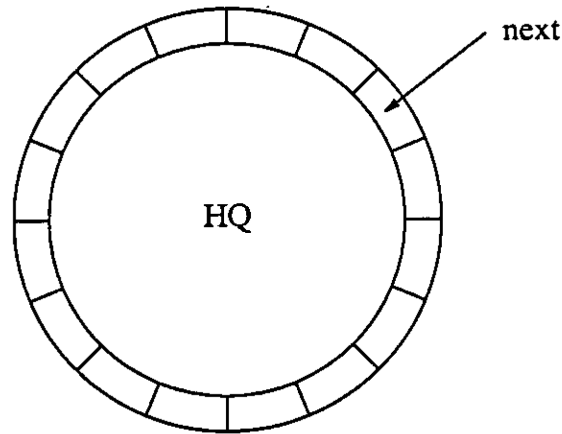


Figure 7.2: Circular History Queue

redo of the last one command. A complete treatment of error recovery and related issues is given in the subsequent section on the state representation system. Examples of policy decisions for bi-directional execution operations (do, undo, redo) regulating state change are:

1. Syntax and availability to users

2. Number and type of commands that can be undone or redone by users

3. If and when to commit commands so that they can not be undone

A design tool system, such as SARA/IDEAS, may exhibit a set of policies to its end users based on the reasoned opinions of a few senior designers, on corporate history, or on the whims of a systems programmer. A tool-building system such as the one presented in this dissertation, SARA-TBS, must not dictate the policy of the tools being built. The tool building system must provide the mechanisms to support a reasonably wide range of policy choices and the ability to specify or to select policy.

Separation of policy from mechanism is a widely supported principle in operating system design, as indicated by the following quotation from Peterson and Silberschatz [Pete83b].

> One very important principle is the separation of *policy* from *mechanism*. ...
>
> The separation of policy and mechanism is very important for flexibility. Policies are likely to change from place to place or time to time. In the worst case, each change in policy would require a change in the underlying mechanism. A general mechanism would be more desirable. A change in policy would then only require redefining certain parameters of the system.
>
> Policy decisions are important for all resource allocation and scheduling problems. Whenever it is necessary to decide whether or not to allocate a resource, a policy decision is being made.

The principle is not often present in operating system implementation as indicated by the following quotation from Brinch Hansen [Hans73].

> For the designer of advanced information systems, a vital requirement of any operating system is that it allow him to change the mode of operation it controls; otherwise, his freedom of design can be seriously limited. Unfortunately, this is precisely what many operating systems do not allow. Most of them are based exclusively on a single mode of operation such as batch processing, spooling, real-time scheduling, or conversational access.
>
> When the need arises, the user often finds it hopeless to modify an operating system that has made rigid assumptions in its basic design about a specific mode of operation. The alternative — to replace the orignial operating system with a new one — is in most computers a serious, if not impossible, matter because the rest of the software is intimately bound to the conventions required by the original system.
>
> This unfortunate situation indicates that the main problem in the design of a multiprogramming system is not to define functions that satisfy specific operating needs, but rather to supply a *system nucleus* that can be extended with new operating systems in an orderly manner. This is the primary objective of the RC 4000 system.

> The basic attitude during the designing was to make no assumptions about the particular medium-term strategy needed to optimize a given type of installation, but to concentrate on the fundamental aspects of the control of an environment consisting of cooperating, concurrent processes.

The UIMS is the system nucleus that Brinch Hansen speaks of.

7.1.2 The Mechanisms

Four mechanisms are identified and elaborated upon: the *state representation mechanism*, the *reference mechanism*, the *interaction mechanism*, and the *input/output mechanism*. Each mechanism is composed of a data structure and controlled access provided by a set of operations as depicted in Figure 7.3.

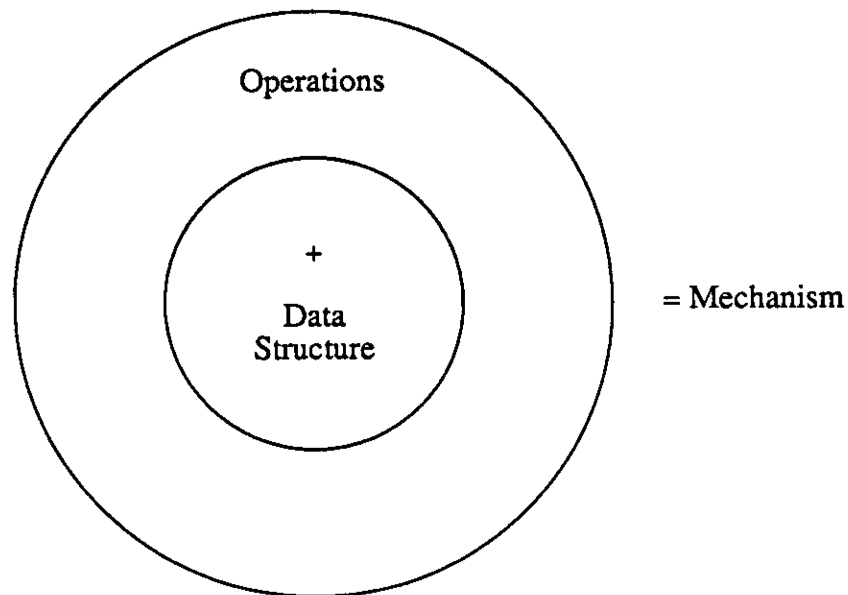


Figure 7.3: A Mechanism Template

The *work space* and the *bi-directional execution operations* comprise the state representation mechanism. A sampling of that mechanism is given in the previous section. The *design data base* and a set of *data base operations* comprise the reference mechanism. The data base is an extension of the work space made necessary by

the limited capacity and the high volatility of the work space.

The interaction mechanism is composed of the *interaction handler* and the *ATN*. The interaction handler is guided by an ATN that captures the syntax and semantics of a specific tool or set of tools. The interaction handler invokes operations upon a dynamic model represented in the work space. All transformations performed on the work space are routed through the bi-directional execution operations. The interaction handler negotiates with the human user through graphical input and output devices.

Finally, the graphical *input/output mechanism* accesses a *display memory* and several components of the objects currently in the work space. The object components used by this mechanism include the *graphics part* of each object, the *putter method* of each object, and the *getter method* of each object.

7.1.3 Chapter Organization

Recall the characterization of an interactive computer system's Model Human User (Human Problem Solving section of chapter 2). The Model Human User is described as three systems, the preceptual system, the motor system, and the cognitive system. Each system is subsequently described as a set of memories and processors. The UIMS Characterization section is developed along similar lines. The chapter is completed by a conclusions section with suggestions for future research.

7.2 UIMS Characterization

The user interface management system is primarily characterized by the *command-response system.* Control of the command-response system resides in an interaction handler. The interaction handler is guided by an augmented transition network (as described in Chapter 5), gets user commands from an input processor, translates user commands into state operations, applies those operations to the set of objects comprising the state, and displays the new state through the services of an output processor.

UIMS characterization begins with a description of the command-response system. Its complex memory hierarchy is discussed first followed by a description of the input processor, the output processor, and the interaction handler. The correspondence between the memories and processors of the command-response system and the memories and processors of the model human user are shown.

The work space memory plays such a crucial role in the command-response system that it is treated separately as the *state representation system.*

The command response system, operating on the state representation system, allows for the efficient creation, modification, analysis, and simulation of designs. In contrast, a design data base is a place in which to store designs in a highly retrievable form. A brief discussion of the *reference system,* including the design data base, concludes the UIMS Characterization section.

7.2.1 Command-Response System

The command-response system is the central component of the UIMS. It is a collection of processors and memories as shown in Figure 7.4. The command-

response system implements the input/output, interaction, and state representation mechanisms.

The interaction handler, guided by the ATN, has a simple two step main cycle.

```
Get next command from user.
Give user appropriate response.
```

The interaction handler gets a user command from the input processor and then responds to it semantically. The semantic response is implemented as a call to an operation in environment memory. If the operation is a transformation operation, then one or more objects in state memory are affected. The display processor is responsible for reflecting the change on a logical output device.

7.2.1.1 Work Space Memory

The work space memory has two major components, the instruction memory and the environment memory. The instruction memory contains the ATN defined in Chapter 5. The environment memory contains, among other things, all modeled objects.

7.2.1.1.1 Instruction Memory

The instruction memory contains the ATN which provides guidance for the interaction handler. The creation of the ATN by the syntax definition compiler is discussed in Chapter 5. The only run-time operation available on the ATN is the selection of the unique start node.

```
package atnPackage
 type atn is private;

 function getStartNode (ATN: in atn) returns node;
  -- an atn has a unique start node.
```

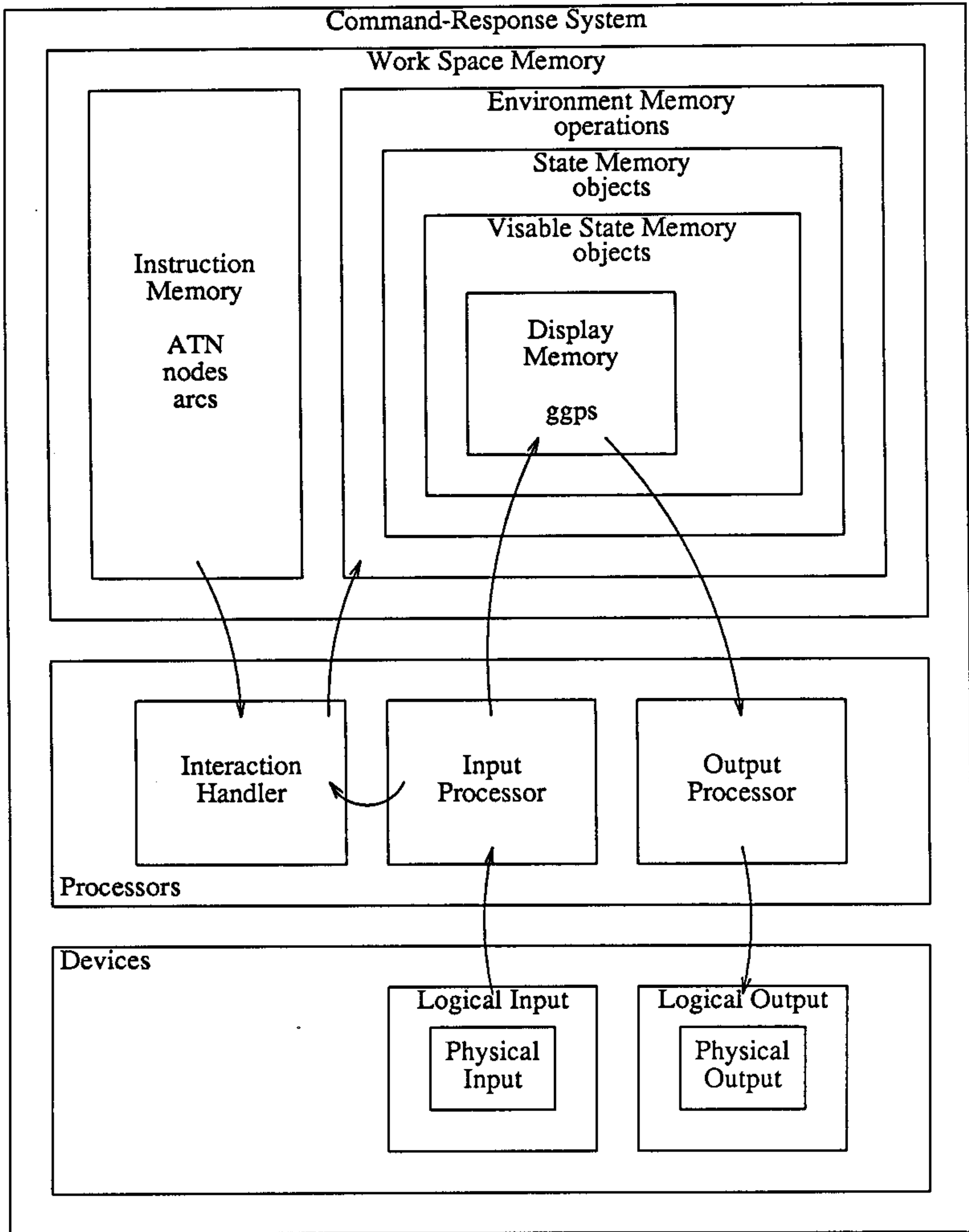


Figure 7.4: Command-Response System

```
end package atnPackage;
```

An ATN is composed of *nodes* and *arcs*.

An ATN node represents a state where the interaction handler comes to rest between interactions with the user. A node may be a start, accept, or intermediate node. Start nodes have no input arcs. Accept nodes have no output arcs. Intermediate nodes have at least one input arc and at least one output arc.

The following operations are defined on nodes.

```
package nodePackage
 type node is private;

 procedure traverseGraph (N: in node);
  -- N is a start node associated with a non-terminal in the grammar.
  -- the procedure returns when it encounters an accept node.

 function isAcceptNode (N: in node) returns boolean;
  -- returns TRUE if there are no output arcs,
  -- FALSE otherwise

 function isStartNode (N: in node) returns boolean;
  -- returns TRUE if there are no input arcs,
  -- FALSE otherwise

 function selectArc (N: in node; T: in terminal) returns arc;
  -- one arc is selected from N's output arc set.
  -- T is either found directly on the output arc,
  -- or it is found in a non-terminal's first set.

 function getNextTerminal (N: in node) returns terminal;
  -- all terminal symbols that can legally be read from this node
  -- are of the same type: category, verb, entity, point, value,
  -- or string. the appropriate feedforward, getter, and feedback
  -- are called. the results of the getter are returned.

 function getter (N: in node) returns terminal;
  -- all terminal symbols that can legally be read from this node
  -- have the same getter method. That common getter method is
  -- executed and the results are returned.

 procedure feedforward (N: in node);
  -- N is prompted for. many level of prompts are possible depending
  -- on a user's skill and the available interaction devices.
```

```
end package nodePackage;
```

An ATN arc represents a transition from one node to another node. A transition from one node to another node is made by selecting one of the current node's output arcs and setting the current node to the destination node of the selected arc. Arcs are labeled with a terminal symbol or a non-terminal symbol from the grammar. A non-terminal symbol has a subgraph associated with it that represents its left- and right-hand side.

The following operations are allowed on arcs.

```
package arcPackage
 type arc is private;

 procedure makeTransition (A: in arc; N: out node);
  -- the operation on the arc is applied to state memory.
  -- the destination node of the arc is assigned to N.

 function hasTerminalSymbol (A: in arc) returns boolean;
  -- if arc is labeled with a TERMINAL, TRUE is returned,
  -- else FALSE is returned.

 function hasNonTerminalSymbol (A: in arc) returns boolean;
  -- if arc is labeled with a NONTERMINAL, TRUE is returned,
  -- else FALSE is returned.

 function getSubgraph (A: in arc) returns node;
  -- arc is labeled with a NONTERMINAL, the subgraph associated
  -- with the non-terminal is returned.

 function applyOperation (A: in arc);
  -- an arc may have an operation that must be called when
  -- a transition is made from the source node to the
  -- destination node.

 function getDestinationNode (A: in arc) returns node;
  -- an arc connects a source and destination node.
  -- the destination node is returned.

end package arcPackage;
```

7.2.1.1.2 Environment Memory

The other component of work space memory is the environment memory. It is analogous to the long term memory of the model human user. Its primary encoding technique is semantic. The environment memory constitutes a mechanism in that it is a collection of operations that provide controlled access to a data structure. The set of operations are those defined during the conceptual definition phase (see Chapter 4), and the data structure is the set of objects also defined during conceptual definition. The set of operations do not change over time, but the collection of objects do. The condition of the set of objects comprises the system state. Therefore, the set of objects are held in a separate partition of the environment memory called the state memory. The operations allowed on the environment memory are do, undo, and redo.

```
package environmentMemoryPackage
 type environmentMemory is private;
 procedure do (C: in command; EM: in out environmentMemory);
 procedure undo (EM: in out environmentMemory);
 procedure redo (EM: in out environmentMemory);
end package environmentMemoryPackage;
```

State Memory

The state memory is a collection of objects entering into relationships with each other as defined during conceptual definition. The objects are not directly accessible except through the operations residing in the environment memory. The operations in environment memory represent the time invariant semantics of the user interface while the objects in state memory represent the time varying semantics of the model being designed. The operations that can be applied to state memory are those

defined in environment memory.

The organization of state memory is so crucial that it is described in detail in a separate, subsequent section.

Visible State Memory

At any one time, only a subset of the objects in state memory are displayed. The visible state memory is that subset of objects in state memory that are curently being displayed. Each of those objects in visible state memory has a graphic-part instance variable and a putter method. The putter method of an object is a routine that draws the shape of that object. Recall that each object was mapped to a shape during syntax definition (see Chapter 5) and a routine was selected from the Device-Independent Shape Library, DISL. The graphic-part is a data structure capable of holding the parameters necessary to draw the shape, e.g., coordinates of a rectangle's corners, or a circle's radius and center point. The graphic-part template is also taken from the DISL. Objects from state memory may be added to or removed from visible state memory.

```
package visibleStateMemoryPackage
 type visibleStateMemory is private;
 procedure addObject(O: in object; vsm: in out visibleStateMemory);
 procedure rmObject(O: in object; vsm: in out visibleStateMemory);
end package visibleStateMemoryPackage;
```

Display Memory

An object's putter method translates its graphic-part into a sequence of Generalized Graphics Primitives, GGPs, and copies the GGPs into display memory. GGPs are discussed in Chapter 6. If an object is represented as a rectangle on a display dev-

ice, then its graphic-part might be a four element list.

graphic-part ≡ (P1, P2, P3, P4)

Its putter method might have a template with one GGP like

drawpolyline (L1)

where L1 is expected to hold a list of points. The putter method would simply copy the template to display memory but substitute the graphic-part for L1, producing a segment in display memory like the following:

drawpolyline (P1, P2, P3, P4)

Four operations can be performed on display memory.

```
package displayMemoryPackage
 type displayMemory is private;

 procedure putter (DM: in out displayMemory; O: in object);
  -- an object's putter method passes in the object and
  -- this procedure extracts its graphic-part and the ggps
  -- from the object's putter. each object so entered in DM
  -- is represented as a segment.

 procedure xyToContainer (P: in point; T: out terminal);
  -- an x,y coordinate is passed in and its closest containing
  -- object is returned as a terminal symbol.

 procedure drawChangedSegments (DM: in out displayMemory);
  -- those segments that have changed since the last call to
  -- this procedure or to drawAllSegments are redrawn.
  -- all segments in DM aremarked as unchanged.

 procedure drawAllSegments (DM: in out displayMemory);
  -- all segments are drawn. All segments in DM are
  -- marked as unchanged.

end package displayMemoryPackage;
```

7.2.1.2 Input Processor

The user's arm-hand-finger movement is the point of contact with the system's physical input devices. Typically, the hand manipulates a keyboard or a mouse. As described in Chapter 6, logical device definitions are interposed between physical devices and the interaction handler.

The input processor is subservient to the interaction handler. The input processor is thus defined to be the implementation of the node operation, function getNextTerminal, and the set of logical input device drivers, getters, selected from the Logical Device Library, LDL, by the device compiler (see Chapter 6). A getter is a software implementation of a logical device on some specific physical device. A getter is one of: select-category, select-verb, select-entity, get-point, get-value, or get-string. Getters beginning with the prefix "select-" are responsible for making calls on xyToContainer to translate coordinates into a terminal that corresponds to the object pointed to.

The following pseudo-code is a simplification of getNextTerminal.

```
function getNextTerminal (N: in node) returns terminal;
  terminal T;

  feedforward(N); -- possibly enable device & prompt.
  T = getter(N);  -- read from the device.
  feedback(T);    -- the object read might have a feedback method
                  -- e.g., the selected menu item might be highlighted.
  return T;
end function getNextTerminal;
```

7.2.1.3 Output Processor

Like the input processor, the output processor is a collection of previously defined operations, a set of display memory operations and the set of putters defined

in Chapter 6.

```
procedure drawChangedSegments(DM: in out displayMemory);
 -- called to update screen.

procedure drawAllSegments(DM: in out displayMemory);
 -- called to totally refresh screen.
```

7.2.1.4 Interaction Handler

The interaction handler is the controlling processor of the UIMS. Its function is to traverse the ATN held in instruction memory. It has six registers, startNode, currentNode, currentArc, currentToken, dollarDollar, and stateStack. After initializing its registers, the interaction handler traverses the ATN.

The interaction handler is described by psuedo-code and also relies on those packages described previously.

```
procedure interactionHandler (G: in ATN);
 register startNode;
 register currentNode;
 register currentArc;
 register currentToken;
 register dollarDollar;
 register stateStack;

begin
 startNode = getStartNode(G);
 currentNode = startNode;
 currentArc = nil;
 currentTerminal = nil;
 dollarDollar = nil;
 stateStack = nil;

 traverseGraph (currentNode);
end;
```

After initializing its internal registers, the traverse procedure is called with the ATN's start state.

Recall from the syntax definition phase that the right-hand side of a syntax rule can be value returning.

```
NT1 : NT2 NT3 NT4
    { (set $$ (sum $1 $2 $3)) }
```

At the time the right-hand side of the rule has been recognized, the values of NT2, NT3, and NT4 have been computed as $1, $2, and $3 respectively. The sum function is called to compute the value of NT1 which is locally called $$. This implies that $1, $2, and $3, must be remembered by the interaction handler at run-time. That is the purpose of the state stack.

The following operations are possible on the state stack.

```
package stateStackPackage
  type stateStack is private;
  type dollarValue is private;

  function push (SS: in out stateStack; V: in dollarValue);
    -- pushes the new V onto the SS. Keeps track of how many
    -- dollarValues have been pushed onto the stack that represent
    -- the right-hand side of the rule currently being parsed.

  function pop (SS: in out stateStack; N: in integer);
    -- N values are destructively read from SS.

  function nth (SS: in out stateStack; N: in integer) returns dollarValue;
    -- the value of $N is non-destructively read from SS.

  function rhsLength(SS: in stateStack) returns integer;
    -- returns the number of symbols in the right-hand side of the
    -- rule currently being parsed.

end package stateStackPackage;
```

The two most important procedures that comprise the interaction handler are the mutually recursive traverseGraph and makeTransition procedures. The following pseudo-code is a simplification of the two procedures.

```
procedure traverseGraph (N: in node);
 if isAcceptNode(N) return;
 currentTerminal = getNextTerminal(N);
 currentArc = selectArc(currentNode,currentTerminal);
 makeTransition (currentArc,currentNode);
end procedure traverseGraph;

procedure makeTransition (A: in arc; N: out node);
 if hasTerminalSymbol(A)
   dollarDollar = applyOperation(A);
   push(stateStack,dollarDollar);
   N = getDestinationNode(A);
   return;
 if hasNonTerminalSymbol(A)
   traverseGraph(getSubgraph(A));
   dollarDollar = applyOperation(A);
   pop(stateStack,rhsLength(N));
   push(stateStack,dollarDollar);
   N = getDestinationNode(A);
   return;
end procedure makeTransition;
```

7.2.2 State Representation System

The state representation system is a memory hierarchy and a set of bi-directional execution operations that provide controlled access to the hierarchy. The memory in question is the work space memory of the UIMS, but more specifically, the portion of work space memory containing objects, the state memory.

7.2.2.1 Purpose

The purpose of this section is to justify the existence of a unique, centralized, state representation system and to identify the operations allowable on state. Several cases of bi-directional execution will be introduced and described, including, but not limited to, user-level error recovery, data base error recovery, version control, and step-mode interactive simulation of designs. A set of bi-directional execution operations is proposed and shown to be a generalization of those cases. A prototype implementation will be presented.

7.2.2.2 Introduction

Interactive systems allow users to construct and modify data objects (documents, programs, models) in real time. Text editors are good examples of this class of interactive program. Other software systems exist for the execution of objects, (data objects or otherwise), e.g., analyzers and simulators. We use the term "analyzer" to mean that class of tool that provides for the execution of a model (data and/or program) statically, like a compiler. We use the term "simulator" to mean the dynamic execution of a model against some collection of data. Simulators are often constructed as interactive programs while analyzers less so.

Editors, analyzers, and simulators share many characteristics, one of which is the ability of the system to move from an initial state towards some final state. In the shadow of this common characteristic many disparate problems can be discussed. To complement forward state transformations, it is desirable to be able to reverse the effects of past actions and to restore the system to a prior state. One way to accomplish this is by saving state in anticipation of restoring to this state. We require an automatic, economic, and convenient solution.

A list of specific examples of forward and backward state transformation follow.

1. The VIsual editor, VI [Joy80a], supports a simple undo. The Chapter 2 section on COPE [Arch84] describes a high quality do/undo/redo facility and introduces the terminology. Many systems have undo added on rather than designed in. Undo is typically added on for user-level error recovery. However, a good undo facility gives rise to bolder user actions, allows automatic systems to exhibit greater initiative, and makes a user more likely to explore

less familiar system features.

2. The C shell history mechanism [Joy80b] provides a redo facility for the command language of the UNIX operating system. It is not integrated with a corresponding undo mechanism.

3. Backward transformation in data base management systems is motivated largely by the need to restore the data base to a legal state after it has entered an incorrect state. The data base may enter an incorrect state because of the actions of an errant or malicious end user, because of electrical power or hardware failure, or because of errors in the implementing software [Date77]. A classic recovery strategy was developed when data bases were stored on magnetic tape. The data base was recorded on one input tape and a sequence of data base transactions were recorded on another input tape. The results of applying the transactions to the data base were recorded on yet another tape. Any error that ocurred during processing could be dealt with by repeating the procedure on the original input tapes. Even if an entire output tape were lost, it could be recreated by repeating the process on a sequence of transaction and data base tapes. A variant of this technique is still used in version control systems.

4. Software development can be characterized as a sequence of versions of a program through its evolution. The source program is a model of a system and the various evolutionary versions are the model's states. Version control systems such as SCCS [Bona77] and RCS [Tich82b, Tich82a] typically bookkeep the steps of development taken but don't provide for reverse execution. Reverse execution is simulated by replaying the change script against the initial state of the model.

5. Execution of models (by simulators and analyzers) is also a sequence of state transformations. Instruction set simulators, such as ISPS [Barb79a, Barb79b, Barb80], typically operate in two modes, step and continuous. Step mode interprets and executes the next instruction and returns control to the user so that the user can examine the contents of simulated machine registers and memory that comprise state. Continuous mode typically executes instructions until an instruction is accessed from a specified address. While executing in continuous mode, users often find themselves attempting to fetch instructions from data space or from uninitialized memory and are unable to ascertain how they got there, i.e., they cannot undo their recent steps.

   Analyzers are typically built as batch tools providing no interactivity at all.

   In both simulators and analyzers, forward execution is well understood. It is the minimum and the norm. Reverse execution is less frequently provided.

6. Automatic programming or automatic design frequently employ forward exploration (perhaps based on blind or heuristic search) and subsequent backtracking. Such techniques rely on the runtime stack built and maintained by the implementation language compiler to implicitly provide for rollback or backtracking. Alternately, the program must explicitly maintain the information for backtracking.

These topics are generally discussed separately even though they are more alike than different. We suggest here that by identifying the salient commonality we can provide a single, general solution to them all. Treated in a unified way, the solution can be economical, convenient, general, and clear.

Central to the discussion is the familiar concept of *state*. State is defined as the set of all system variables and their values. Given that definition and the terminology presented in the next section, we are ready to build the first level of our solution based on *state change* and *state restore*. Several forms of state representation are given. Economy of the chosen form is shown by comparing it with other forms of *state capture*. After the form of state representation is presented, the set of state operations, through which all state change requests must be channeled, is presented.

7.2.2.3 Terminology

This section introduces the terminology necessary to discuss bi-directional execution.

WS : WorkSpace, an area containing all objects that may change during system execution.

IH : the Interaction Handler, a processor that submits requests on behalf of the user to transform objects within a WS.

T_m : some state Transforming operation.

S_n : some State.

O : some set of Objects.

O_i : some Object.

A : some set of Attributes.

A_j : some Attribute.

V : some set of Values.

V_k : some Value.

SE : a Script Element composed of a single transformation operation and a specification of the object's attributes that are to receive new values, i.e., $T_m(O, A, V)$.

EX : that portion of the interaction script already EXecuted.

PX : that portion of the interaction script Pending eXecution.

Throughout this section, a rectangular grid, such as the one shown in Figure 7.5, is used to represent a snapshot of system state. The rectangle is labeled S_n where n is the number of state transformations that have been made and that are still in effect since the session began. State is composed of many objects which are in turn composed of many attributes. Objects are shown along the horizontal axis and their attributes are shown along the vertical axis.

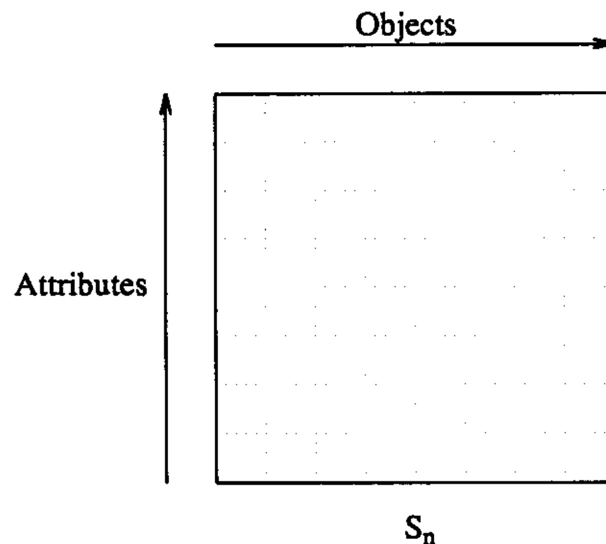


Figure 7.5: State Representation Template

The only operations performable on state are do, undo, and redo. Their semantics are described by a combination of state transformations and script operations. User commands undergo a translation into requests for transformation opera-

tions. These transformations are represented as script elements in PX and EX.

In order to perform a state transformation, a representation of the command and its arguments is pushed onto the front of PX, the transformation operation is applied to state, and the command representation is moved from PX to EX.

$$\text{DO } T_m \,(O,A,V) \Rightarrow \text{PUSH } (T_m(O,A,V)\,,\, PX)$$

$$\Rightarrow (T_m\,(S_n)) \rightarrow (S_{n+1})$$

$$\Rightarrow \text{PUSH } (\text{ POP } (PX)\,,\, EX)$$

Undoing an operation is described as restoring the current state from S_n to its former value S_{n-1}, followed by moving the command representation of T_m from EX to PX where it can be redone if necessary.

$$\text{UNDO } T_m \Rightarrow (T_m^{-1}\,(S_{n+1})) \rightarrow (S_n)$$

$$\Rightarrow \text{PUSH } (\text{ POP } (EX)\,,\, PX)$$

To redo a transformation, the top most command representation from PX is applied to state and then is moved to EX.

$$\text{REDO } T_m \Rightarrow (T_m\,(S_n)) \rightarrow (S_{n+1})$$

$$\Rightarrow \text{PUSH } (\text{ POP } (PX)\,,\, EX)$$

A possible fourth operation on state is *commit*. Its effect is to render all commands on EX undoable. It is performed by forgetting all attribute values but those values current in state S_n.

$$\text{COMMIT} \Rightarrow n \leftarrow 0 \quad \text{i.e., } S_n \text{ is } S_0$$

=> FLUSH (EX)

7.2.2.4 Computational Inverse Functions

One way to provide forward and backward state transformations is to write a function that computes the inverse of each forward transformation function. This approach to state representation and do/undo/redo requires the tool developer to define the transformation functions that take the system from state S_i to state S_{i+1} *and* inverses of those transformation functions that take the system from state S_{i+1} to state S_i.

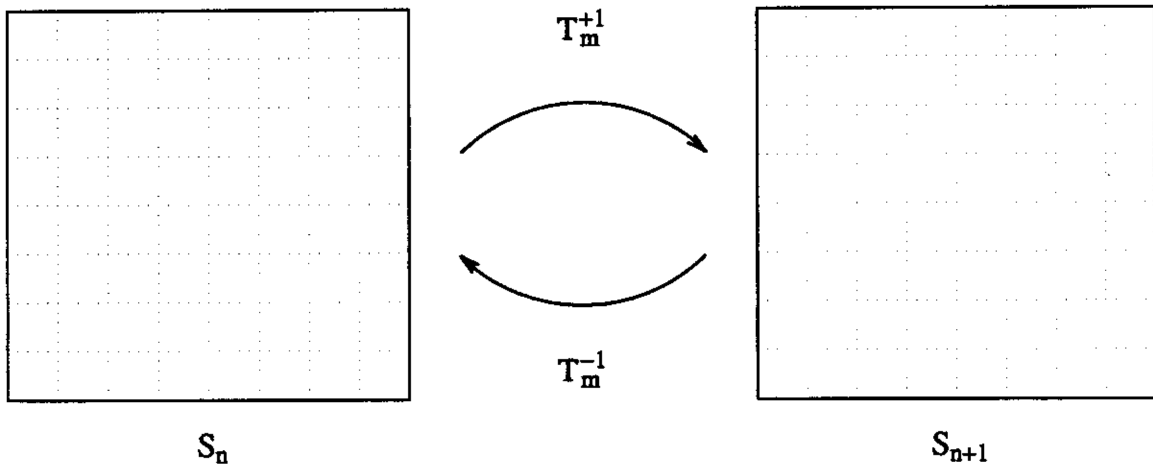


Figure 7.6: Computational Inverse Functions

For example, consider the simple transformation, increment.

$$T_{inc}\ (X) \equiv X \leftarrow X+1$$

A simple inverse exists, decrement.

$$T_{inc}^{-1}\ (X) \equiv T_{dec}\ (X) \equiv X \leftarrow X-1$$

The tool implementor must write both functions and somehow bind them together as inverses. Whenever it is necessary to undo an operation, its inverse is found and

executed. However, consider another simple transformation, square, raise to the power of two.

$$T_{SQUARE}\ (X) \equiv X \leftarrow X^2$$

When it becomes necessary to undo T_{SQUARE}, the former value of X has been overwritten and is not available. In order to restore the previous value of X, the computational inverse function, $T_{SQUARE} = T_{SQRT}$, is called and can only produce both a positive and a negative root of X. The correct root is impossible to determine unless care has been taken to save the old value.

Not all functions have purely computational inverses. The fundamental difficulty is the information loss associated with many obvious and non-contrived transformations. Computational inverse functions are considered here only to motivate the need for additional memory. The next three methods all support both forward and backward state transformations by exploiting extra memory.

7.2.2.5 Not-So-Instant Replay

One strategy to provide bi-directional execution can be called not-so-instant replay. The fundamental notion is to keep the state present at the beginning of the session, the original state, intact. A copy of the original state is made and called the current state. Each transformation takes the current state into the next state which then replaces the current state. In order to undo the last command, the original state is again copied to the current state and the entire session transcript, save the last script element, is replayed.

For example, a session might progress from S_0 to S_z through some sequence of transformations (T_1, T_2, ..., T_{z-1}, T_z).

$$S_1 \leftarrow T_1(S_0, O, A, V)$$

$$S_2 \leftarrow T_2(S_1, O, A, V)$$

$$\cdots$$

$$S_{Z-1} \leftarrow T_{z-1}(S_{Z-2}, O, A, V)$$

$$S_Z \leftarrow T_z(S_{Z-1}, O, A, V)$$

When the user requests a rollback from S_z to S_{z-1} the system first restores the current state to S_0 and then replays all but the last transformation, i.e., T_{z-1} but not T_z.

$$S_1 \leftarrow T_1(S_0, O, A, V)$$

$$S_2 \leftarrow T_2(S_1, O, A, V)$$

$$\cdots$$

$$S_{Z-1} \leftarrow T_{z-1}(S_{Z-1}, O, A, V)$$

This implementation strategy is intolerably slow if the number of transformations per session is high and the need for backward execution is relatively high. However, it may be appropriate if memory space is at a premium and backward execution is rare. Version control software typically employ this strategy with success.

7.2.2.6 Complete State Capture

The need for fast backward execution may obviate the use of not-so-instant replay and justify the space costs of the complete state capture strategy. Initially, the original state is called the current state. Each transformation is applied to the current

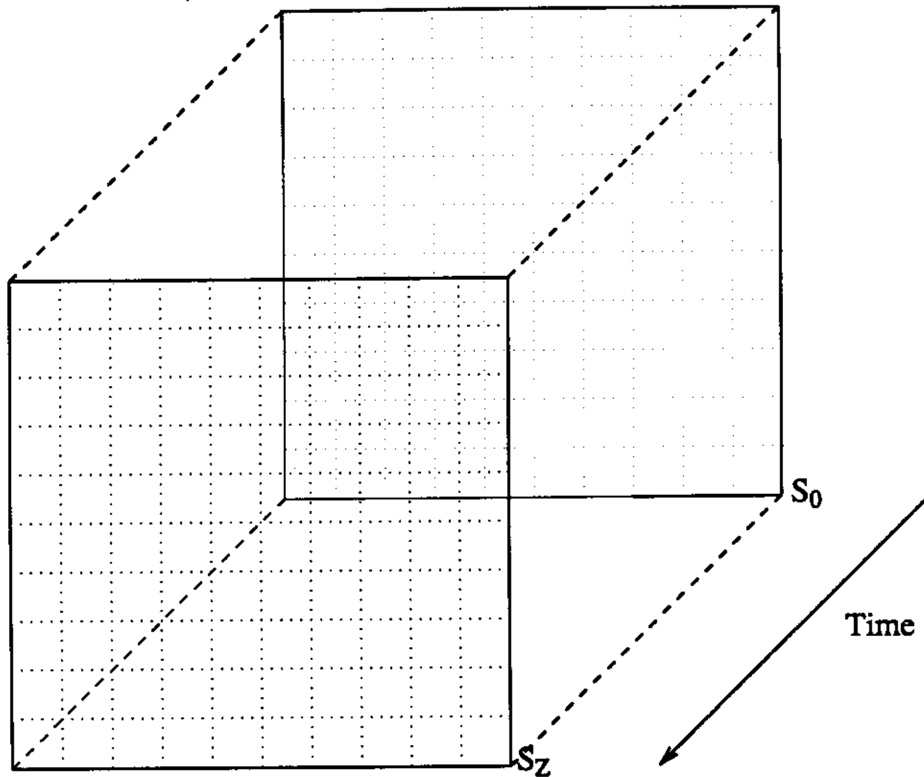


Figure 7.7: Not-So-Instant Replay

state producing a new state. The new state is called the current state, but, unlike not-so-instant replay, the old current state is retained. Thus if Z transformations have been applied to the initial state,

$$S_1 \leftarrow T_1(S_0, O, A, V)$$

$$S_2 \leftarrow T_2(S_1, O, A, V)$$

$$\ldots$$

$$S_Z \leftarrow T_Z(S_{Z-1}, O, A, V)$$

then there will be Z + 1 states consuming memory.

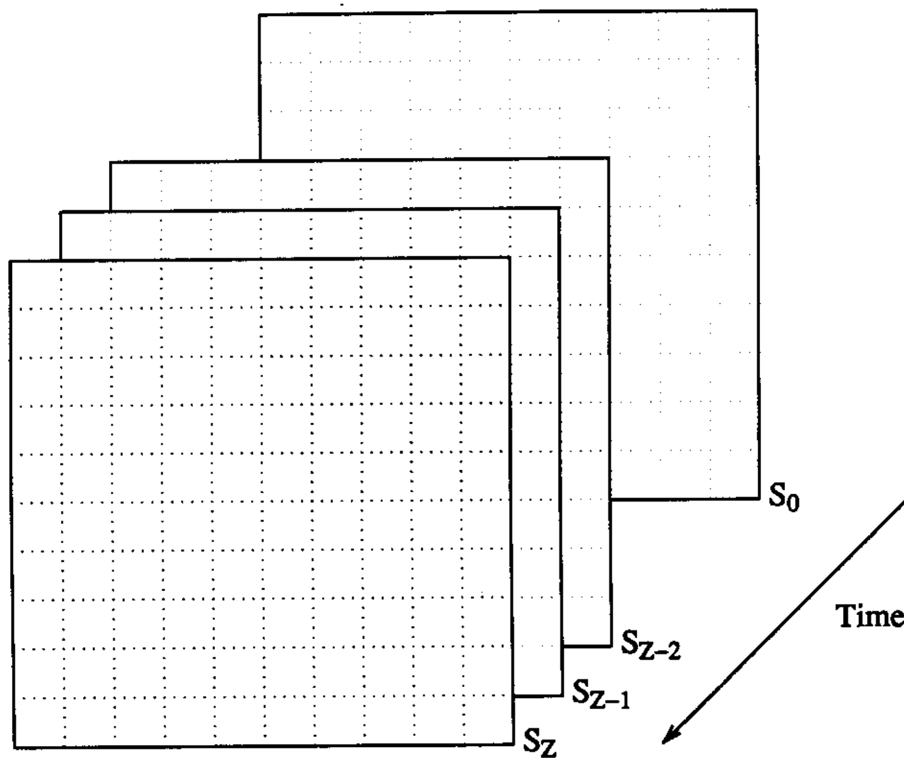


Figure 7.8: Complete State Capture

The space consumed by the complete state capture strategy is large and proportional to session length. It does, however, solve the time problem presented by not-so-instant replay.

Automated Teller Machines, ATM, interact with extremely unsophisticated and error-prone users and with extremely large data bases with high reliability requirements. ATM sessions are typically short and the state of an individual user's account is quite small compared to the entire data base. Complete state capture is a reasonable strategy for this application.

7.2.2.7 Minimal State Capture

The bi-directional execution strategy chosen for the SARA-UIMS strikes an effective compromise in space and time requirements. The minimal state capture strategy exploits the fact that during any single transformation operation, the entire state does not change. Nor do entire objects change necessarily. Typically, only a few attributes of a few objects change and only the changes in those few values must be recorded. A picture of Z transformations applied to the initial state, S_0, is shown Figure 7.9.

$$S_1 \leftarrow T_1(S_0, O, A, V)$$

$$S_2 \leftarrow T_2(S_1, O, A, V)$$

$$\ldots$$

$$S_Z \leftarrow T_Z(S_{Z-1}, O, A, V)$$

Minimal state capture offers a reasonable compromise when compared to not-so-instant replay and complete state capture. A complete comparison of the methods is given in the next section.

7.2.2.8 Comparison

This section quantitatively compares the various state capture methods. The space and time comparision is summarized in Figure 7.10. The computational inverse function approach failed to meet semantic requirements, but is included in the table as a point of departure.

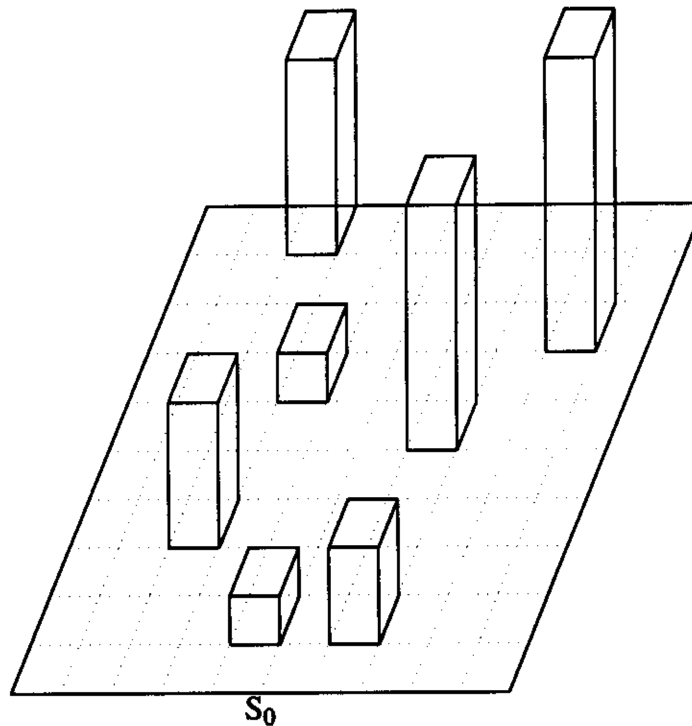


Figure 7.9: Minimal State Capture

Three types of variables are used, typical session quantities, time costs, and space costs.

Session Quantities

N_T: average number of transactions in session

$N_{A/T}$: average number of attributes affected per transaction

N_O: average number of objects comprising state

$N_{A/O}$: average number of attributes per object

Space costs

S_{SE}: space for script element

S_A: space for attribute

S_O: space for object, $S_O = S_A \times N_{A/O}$

S_S: space for entire state, $S_S = S_O \times N_O$

Time costs

T_T: average transformation time

T_{AA}: average time to allocate S_A

T_{AA}^{-1}: average time to deallocate S_A

T_{SE}: time to allocate S_{SE}

The space and time costs of script maintenance, S_{SE} and T_{SE}, are not included in the cost calculations since they are small and common to all methods.

Space and Time Comparison of State Capture Methods			
Method	Space	Forward Step	Backward Step
Computational Inv	S_S	T_T	T_T
N-S-I Replay	$2 \times S_S$	T_T	$(\frac{N_T}{2} - 1)T_T$
Complete State	$N_T \times S_S$	$T_T + T_{AA} \times N_O \times N_{A/O}$	$N_O \times N_{A/O} \times T_{AA}^{-1}$
Minimum State	$S_S + N_T \times N_{A/T} \times S_A$	$T_T + T_{AA} \times N_{A/T}$	$N_{A/T} \times T_{AA}^{-1}$

Figure 7.10: Space and Time Comparison of State Capture Methods

Space Consumption

Not-so-instant replay requires only marginally more space than inverse functions. Most importantly, it is completely independent of session length. Also noteworthy is the fact that as the amount of space to hold all of state increases, i.e., as the number of objects increases, space consumption increases only by a factor of two.

The space requirement of complete state capture is extremely sensitive to long sessions and to large numbers of objects.

The space requirement of minimal state capture is proportional to session length like complete state capture. This dependency is mitigated by the relatively few attributes whose new values require the allocation of space, for typical operations.

Forward Step Time

The two most important parameters of forward step time are the time required to apply the transformation, T_T, and the time required to allocate an attribute, T_{AA}. The time required to perform a forward transformation for all state capture methods is independent of session length. Complete state capture and minimum state capture are sensitive to the number of objects comprising state. The dependency exists because both methods must dynamically allocate storage space during forward transformations. Not-so-instant replay is insensitive to the number of objects since the same space is reused. If T_T is overshadowed by T_{AA}, various memory allocation strategies may be applied to reduce T_{AA}.

Backward Step Time

The backward transformation time of not-so-instant replay is severly dependent on the session length and the cost of reapplying the transformations must be paid again. The most important time cost associated with complete state capture and minimum state capture is the time to deallocate the space of one attribute. In a practical implementation, this cost would not be paid at the time a backward transformation was made but at some later time during garbage collection. Complete state capture is sensitive to the number of objects comprising state and spends significant

time deallocating space that did not need to be allocated in the first place. Minimum state capture is most dependent on the few number of attributes that actually change during a transformation.

Summary

Minimum state capture represents a time and space cost effective solution to bi-directional execution. For low values of $N_{A/T}$ it is only modestly dependent on session length, N_T, and on object population, N_O.

7.2.2.9 The Mechanism

This section describes an implementation of minimal state capture. Not shown is the interaction handler's translation of user requests into calls on the bi-directional execution operations, do, undo, and redo. Four T objects are presented, the work space, the script, the script element, and the attribute. The portions of those objects germain to the state representation mechanism are discussed in turn. After each T object has been discussed, a **do** operation will be followed through the sequence of operations necessary to perform state capture.

As a convention, the instance variables whose changes are not monitored end in "-part" and instance variables whose changes are monitored end in "-attribute".

For this discussion, the most important part of the work space is the script-part and the methods that operate on it. The most important methods are script-append, script-truncate, and script-reappend. A fourth method, delta, will be described during the hand execution of **do**.

7.2.2.9.1 The Work Space Object

The script operations on the work space result in the same operation being invoked on the work space's script-part.

```
(define (MakeWorkspace)
  (let ((...)
        (script-part (MakeScript)))
      (object nil
          ((...))
          ((script self) script-part)
          ((script-append self cmd) (script-append script-part cmd) self)
          ((script-truncate self) (script-truncate script-part) self)
          ((script-reappend self) (script-reappend script-part) self)
          ((delta self attr) (delta script-part attr))
          (((setter script) self val) (set script-part val))
          ((undoable? self) (undoable? script-part))
          ((redoable? self) (redoable? script-part))
          ((workspace? self) t))))
```

7.2.2.9.2 The Script Object

The script has two important instance variables, ex-part and px-part. Ex-part holds script elements of transformations already executed and px-part holds those script elements pending execution. A third instance variable, tmp-se-part, is a temporary variable that holds a new script element while it is being filled in.

The script-append and script-reappend methods construct a script element to document the transformation currently underway and call the T eval function to affect the transformation. The script-truncate method also performs bookkeeping tasks on the script element, but instead of evaluating a command, it calls for a roll-back on the script element that is being removed from px-part. The script's delta method, as did the work space's delta method, just passes its duties onto the T object below it in the hierarchy. The T object below the script is the script element.

```
(define (MakeScript)
  (let ((ex-part nil) ; part of script already ex-executed
```

```
    (px-part nil) ; part of script pending ex-execution
    (tmp-se-part (MakeScriptElement)))
   (object nil
      ((...))
      ((executed self) ex-part)
      ((pending self) px-part)
      ((delta self attr) (delta tmp-se-part attr))
      ((script-append self cmd)
       (tmp-se-part tmp-se-part (MakeScriptElement))
       (tmp-se-part (command tmp-se-part) cmd)
       (eval cmd *scratch-env*)
        ; call the xfrm, it will call delta as needed
       (push ex-part tmp-se-part)
       self)
      ((script-truncate self)
       (cond ((null? ex-part) nil)
           (else
            (set tmp-se-part (pop ex-part))
            (rollback tmp-se-part)
            (push px-part tmp-se-part)
            self)))
      ((script-reappend self)
       (cond ((null? px-part) nil)
           (else
            (set tmp-se-part (pop px-part))
            (eval (command tmp-se-part) *scratch-env*)
            (push ex-part tmp-se-part)
            self)))
      ((undoable? self) (not (null? ex-part)))
      ((redoable? self) (not (null? px-part)))
      ((script? self) t))))
```

7.2.2.9.3 The Script Element Object

A script element has two instance variables, a command-part and an affected-attributes-part. The command-part holds a representation of a transformation operation and the operation's arguments. The affected-attributes-part holds a list of those object's attributes that are affected by this operation.

There are methods to set and retrieve the command-part of a script element, and there is a rollback method that is invoked by the script-truncate method of the script. The script element's rollback method applies the attribute's rollback method

to each element of the list of affected attributes. In addition, there is finally a delta method that does little more than just punt to a subordinate object.

```
(define (MakeScriptElement)
 (let ((command-part nil)
       (affected-attributes-part nil))
   (object nil
         ((...))
         ((command self) command-part)
         ((rollback self)
          (iterate search ((attr (car affected-attributes-part))
                          (rest-of-attrs (cdr affected-attributes-part)))
                (cond ((null? attr)
                      (set affected-attributes-part nil)
                      self)
                     (else
                      (rollback attr)
                      (search (car rest-of-attrs)
                           (cdr rest-of-attrs))))))
         (((setter command) self val) (set command-part val))
         ((delta self attr) (push affected-attributes-part attr))
         ((script-element? self) t))))
```

7.2.2.9.4 The Attribute Object

Each attribute has a present value and a list of past values. The rollback method overwrites the present value with the most recent value from the list of past values and removes that value from the list. The value-part is maintained as a stack with the top value being the present value.

The other interesting attribute methods are the ones to read and write the value of an attribute, the current value, cv, methods. An attribute's value cannot be examined or altered directly. When the value of an attribute is requested, cv provides the present value. When an attempt is made to alter an attribute, the new value is pushed onto the value-part.

The controlled access to all attributes through the cv method constitutes the lowest level mechanism in the SARA-UIMS. Recall that the semantic phase of SARA-TBS generated the T object equivalent of each object that could be manipulated via the user interface. During the generation of T objects, the OReO compiler must insert calls to the cv method.

```
(define (MakeAttribute attr-name)
 (let ((value-part nil))
    (object nil
        ((...))
        ((cv self) (car value-part))
        ((rollback self) (pop value-part) self)
        (((setter cv) self newValue)
         (push value-part newValue)
         (delta *ws* self)
         self)
        ((attribute? self) t))))
```

7.2.2.9.5 Walkthrough of the Do Operation

We now hand execute a **do** operation through the sequence of T objects just introduced. The mysterious delta method will be described. The example begins after the interaction handler translates a request to **do** a command into a script-append operation on the work space with the command as argument.

1. The work space method for script-append simply calls script-append on the work space's script-part.

2. The script's script-append method performs four steps.

    a. a new script element, se, is created.

    b. the command-part of the script element is set to the command (actually performed by a script element method).

c. the command is actually applied to its arguments.

d. the command is entered into the executed part of the script.

3. As the command is applied to its arguments, attributes are changed. This causes the attribute's cv setter to be invoked. If many attributes are changed by a single transformation, then there are correspondingly many calls on the cv setter. The cv setter is implemented in two steps.

    a. the new value of the attribute is pushed onto the value-part.

    b. the delta method is called on the work space with the attribute as argument.

4. The work space's delta method passes control to the script's delta method.

5. The script's delta method passes control to the script element's delta method. The script element in question is the one held in the temporary instance variable, se, which was allocated in step 2a.

6. The script element's delta method adds the changing attribute to its list of affected attributes. If N attributes are changed by one transformation operation, then there will be N calls on delta and N attributes in the script element's list of affected attributes.

7.2.2.9.6 Mechanism Summary

The UIMS kernel must support *mechanism* not *policy*. A tool designer using the SARA-TBS may wish to impose a variety of policies on the end user and may not wish to pay the space and time cost of a state representation mechanism as just suggested. Alternate policies can be supported by implementing families of script

routines. By identifying several alternative interactive models with various degrees of recovery and implementing a set of cooperating script routines for each alternative we can achieve a balance of cost and performance.

7.2.3 Reference System

The reference system is composed of a large design data base and an association processor. Previously solved designs are stored in a fashion that allows them to carry associations with other designs. The association processor allows the design data base to be traversed and examined according to associations with other designs or known quantities. The association processor has been called a browser [Land83] in other SARA research.

Landis [Land86] proposes a relational data base that integrates with the SARA-TBS and the SARA-UIMS.

Because the types of relations that might exist between designs is unpredictable, it is recommended that each tool that results in a storable design should have a separate subdialogue written for it that allows the end user to browse through the design data base.

7.3 Conclusions and Suggestions for Further Research

The implementation of the user interface management system meets its stated goals. It directly supports the interaction needs of its users by providing a well chosen set of policy-independent mechanisms integrated with SARA-TBS. The implementation strategy is shown to be efficient and cost effective.

The bi-directional execution operations promote more liberal exploration. The user is free to explore sequences of operations without fear that his or her steps cannot be retraced. If the art of design were better understood, the rules known to a successful designer could be captured in an expert system's corpus of rules. This control knowledge could be offered by the expert system, acting as a surrogate for a human user, interacting with the interaction handler.

But what is design? A more powerful family of script operations can be implemented that would allow the human to document why certain design branches were explored, why some design alternatives, once explored, were rejected, and why others were selected. A full design audit trail can be captured. This audit trail could form the basis for research into how successful designers approach the art of design.

CHAPTER 8

SARA/IDEAS - A Computer-Based System Design Methodology

System ARchitect's Apprentice, SARA, is a requirement-driven top-down and bottom-up design method for concurrent digital systems [Camp78]. SARA supports the design process of complex concurrent digital systems. Both hardware and software design are supported. The SARA method is supported by an extensive body of software designed at UCLA and implemented in the PL/1 language on the MULTICS system at MIT. It provides separate tools for the structural and the behavioral modeling of systems. The history of SARA is given in [Estr78].

This chapter first introduces the SARA design methodology and then describes, in separate sections, each major tool or subsystem that makes up the SARA tool system. Concurrent reader and writer processes communicating through a shared buffer are employed as a running example to clarify the design procedure and its supporting tools.

8.1 Design Procedure

This section describes the SARA design procedure as depicted in Figure 8.1. The design process is *initialized* by insisting that the designer partition the universe of design discourse into a *system module*, and an *environment module* in which the system will operate. This first step may seem rather mechanical, but its omission is the cause of many faulty designs. The environment module is made explicit so that the designer is forced to focus attention on what assumptions are being made about it. Once those assumptions are documented, the designer turns attention to specifying requirements to be met by the system module.

Neither the environment assumptions nor the system requirements are supported by a formal language and a corresponding language analyzer. Although Winchester [Winc81] has proposed a SARA requirements definition language and a requirements analysis technique, it has not been implemented as a SARA tool.

The next step in initialization is the development of a high-level behavioral description of the system module. Behavior is described in three different domains, the control flow, the data flow, and the interpretation domains. Each is supported by a language and language analyzer. Collectively, they are supported by a simulator.

Both *top-down partitioning* and *bottom-up composition* are supported. If the system being designed is simple it may be described immediately in the three languages already mentioned.

Designers are rarely faced with the task of starting from scratch. Complex systems are composed of many subsystems and it may be possible to re-use what another designer has already provided. A power supply is an obvious example of a re-usable subsystem.

If, for example, the system being defined is a variant on a well established product line, it may be possible to search an existing library of previously designed and tested modules. These *building blocks* can be collected to form a *composition*. If the product line is special-purpose digital controllers, the building block library might contain descriptions of TTL DIP chips that typically comprise major portions of the product line.

However, a system is likely to require the design of a new subsystem or component. If the new subsystem is large, divide-and-conquer is employed. The system module is *partitioned* into smaller, more manageable modules. Initialization is repeated for each new module thus identified. Each new module becomes a system that exists within a containing environment.

Regardless of the tactic taken, partitioning or composing, the resultant design is tested using the many tools in the SARA complement. These tools are generally one of two types, analyzers or simulators. The results of analysis or simulation are tested against the requirements. If requirements are met, then the designer may turn attention to another module. If requirements are not met, a new partition or composition is attempted in a search for satisfaction of requirements.

In the following sections, each major step in the methodology will be discussed in greater detail using the reader-writer example.

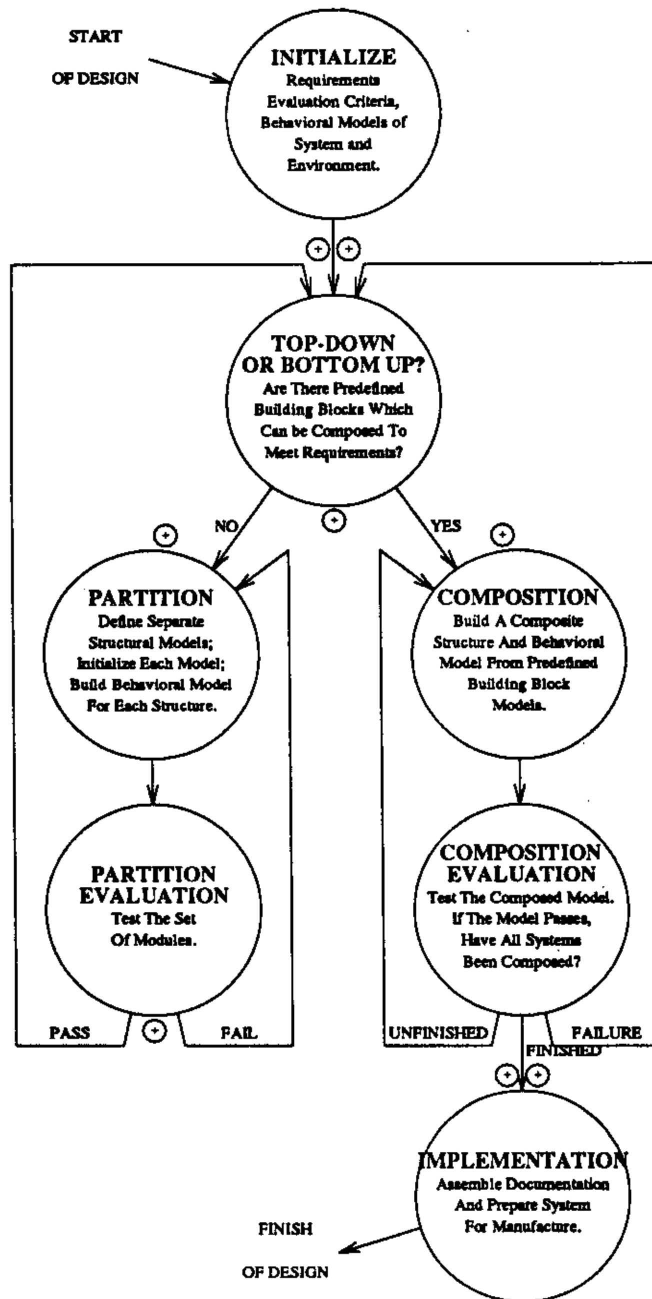


Figure 8.1: The SARA Methodology

8.2 Requirements Definition

The SARA methodology is requirement-driven, yet it has no supported requirements definition language nor language analyzer. Winchester [Winc81] has proposed such a language and has defined a set of analysis techniques, tools, and procedures that fill this requirements definition subsystem gap.

The Requirements Definition Language, RDL, is used to separately specify the functional, process, and attribute requirements that comprise the semantic model of the computer system being specified. The semantic model is composed of six primitives that describe the structural and behavioral components of the system. Winchester describes a correspondence between these six primitives and those of the extant SARA system. Given this correspondence, it is possible to generate SARA models from RDL and to apply SARA analytical and simulation tools in the verification of the specification.

In lieu of an RDL specification, the next two sections describe the environment assumptions and the first definition of the system module for the reader-writer example.

8.2.1 Environment Assumptions

The environment will contain two processes: **`sender`** and **`receiver`**. The **`sender`** process behaves as follows:

- It sends messages to the buffer system through the **`write`** procedure.
- It sends only one message at a time and, after sending one message, can proceed only after **`write`** finishes.

- After the last message is sent, **sender** terminates.

The **receiver** process behaves as follows:

- It requests messages fom the buffer system through the **read** procedure.
- It reads one message at a time and, after requesting a message, can proceed only after **read** finishes.
- After the last message is read, **receiver** terminates.

8.2.2 System Module

```
with SARA_INTERFACE; use SARA_INTERFACE;
package buffer_package is
  type message_slot is private;
  type buffer is array(1..MAX) of message_slot;
  procedure write(m : in message_slot);
  function read returns message_slot;
  procedure init;
end buffer_package;
```

A buffer is, then, a sequence of MAX message_slots. procedure init initializes the buffer to be empty. Calls to procedures read and write can occur concurrently. If write is called when the buffer is full, the caller will be inhibited until there is an empty slot in the buffer. If read is called when the buffer is empty, the call will be inhibited until some message is written onto the buffer. The system needs to manage the buffer in such a way to ensure that:

- Messages are delivered in the same order as they are received.
- No message is destroyed (written over) before being read.
- No message is read twice.

8.3 Structural Modeling

The structure of a system is expressed in terms of the Structure Model, SM. The SM has three primitives: *modules, sockets* and *interconnections*. Modules can be connected with other modules by an interconnection connecting two sockets, one socket in one module and one socket in the other module. Thus, sockets are communication ports for modules.

The interconnection is not directed, it just models a communication line but does not reveal which way the information flows. An interconnection always connects two and only two sockets. Furthermore, a socket can have only two interconnections attached to it: one going out and one coming in. Hierarchical decomposition is achieved by refining a module into submodules.

There is a top level module called universe which has no sockets. Hierarchical decomposition is achieved by refining a module into submodules. This process can be repeated until the system has been decomposed into small enough modules, whose behavior can be directly mapped to an exisiting behavioral model stored in the Building Block Library or whose behavior is simple enough to be understood and expressed using the behavioral primitives.

In our buffer system, we would decompose our universe module into the buffer system and its environment. The environment and the buffer system would communicate through the write and read operations. Figure 8.2 shows the SM for

the Buffer System.

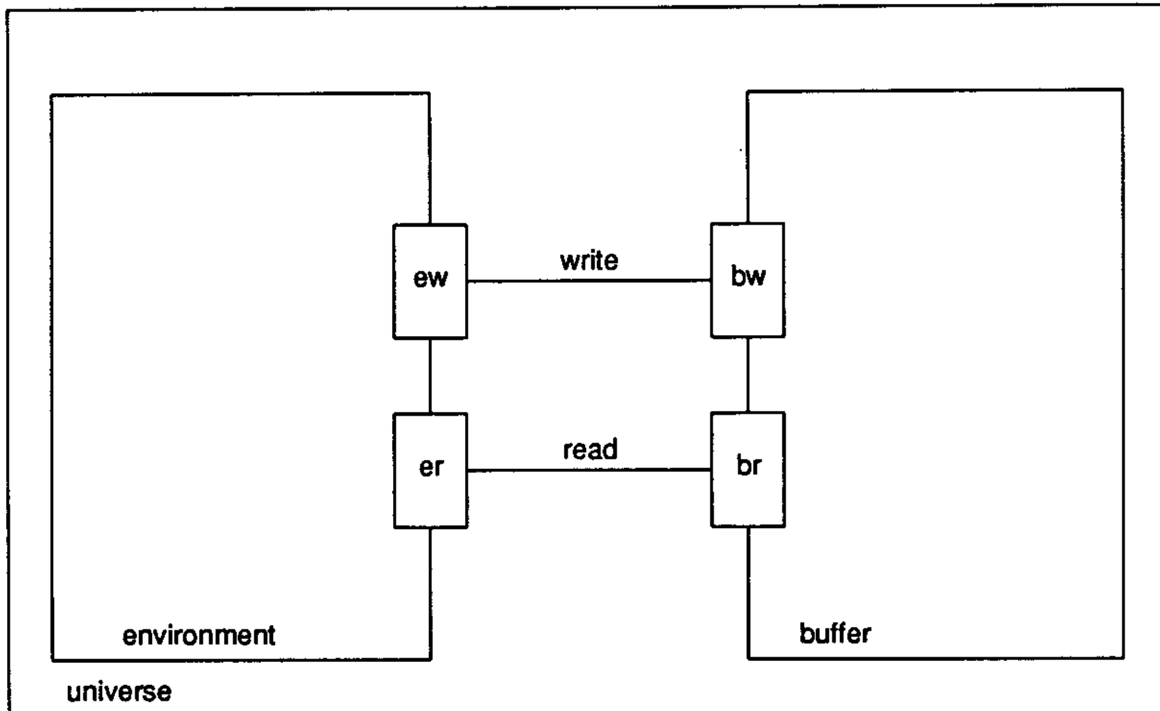


Figure 8.2: Structure Model of the Buffer System

The environment module has two sockets: ew (for environment write) and er (for environment read). These sockets are connected through interconnections write and read respectively to sockets bw and br in the buffer module.

The environment module or the buffer module could be partitioned further into submodules if needed.

8.4 Behavioral Modeling

In SARA, the behavior of the system is modeled using the Graph Model of Behavior, GMB [Razo77]. The GMB offers the designer three different but related modeling domains, control, data, and interpretation. The designer focuses on one of

these domains at a time. After developing independent systems descriptions in each domain, the designer insures that they are consistent with each other.

8.4.1 The Control Domain

The control flow model describes concurrency, synchronization and precedence relations in a graph using an underlying theoretical model similar to Petri-Nets [Pete81].

The control domain of the GMB is a directed hypergraph, i.e., a graph in which the edges may have multiple sources and/or multiple destinations. *Control nodes* (the vertices) represent events and *control arcs* represent precedence constraints, or a partial ordering, among the events.

Each node has an *input logic expression,* which is a boolean expression on the input arcs, that expresses the condition under which that node can be initiated. An OR, ''+'', in the input logic means any of the operand arcs can initiate the node. An AND, ''*'', in the input logic means that all operand arcs must pass control before that node can be initiated.

Each node has an *output logic expression,* a boolean expression on the output arcs, which shows where control is passed upon termination of that node. An OR here implies control is passed to one of the designated arcs. An AND implies control is passed to all of the designated arcs.

Both input and output logic expressions can be arbitrary functions using ANDs and ORs. Control flow in the control graph is represented by the passing of *tokens* through control arcs. When a node is initiated, it consumes the tokens which enabled it. Upon node termination, tokens are created and placed on output arcs

according to the node's output logic expression. The semantics of a control graph are dictated by an underlying machine known as the *token machine* which performs state-to-state transformations on the graph, starting from an *initial token distribution* and terminating if and when no further transformations are possible.

Continuing with our buffer system, we define a control graph for each of the modules defined in the SM. To show this mapping, each control graph can be drawn on its corresponding SM module:

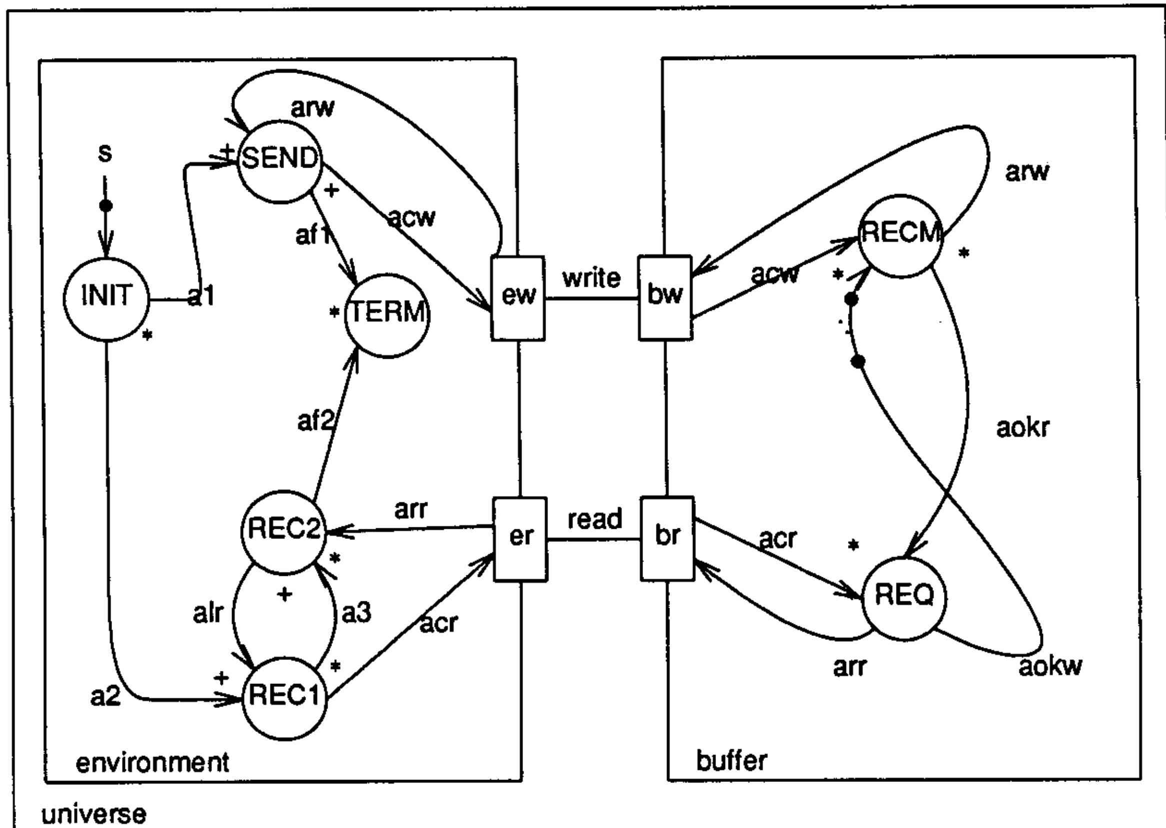


Figure 8.3: GMB Control Graph for the Buffer System

The following tables describe the function of the various components in the control graph:

Control Nodes

INIT	Initiation process, initiates sender.
TERM	Termination process.
SEND	Sender process, sends message to the buffer and receives acknowledgment.
REC1	Receiver process 1, requests message.
REC2	Receiver process 2, actually receives message from the buffer and informs REC1.
RECM	Receives message from the environment, acknowledges, and performs the write operation.
REQ	Receives request, performs the read operation, and sends it to the environment.

Control Arcs

s	Arc to initiate the system.
aokw	Semaphore, indicates the number of empty slots in the buffer.
aokr	Semaphore, indicates the number of messages in the buffer.

8.4.2 The Data Domain

The data domain of the GMB is a bipartite directed graph, i.e., a graph in which there are two kinds of nodes (*datasets*, represented as rectangles and *data processors*, represented as hexagons) and in which arcs (called *data arcs*) are used to connect datasets with data processors. Thus, every data arc goes from a data processor to a dataset or viceversa. This graph represents the data flow of the system by defining its data paths.

Datasets model static collections of data. Data processors are data transformers which can read from and/or write to datasets. Data arcs define the read and write accesses of a data processor to a dataset.

Continuing with the buffer example, we draw the data graph over the SM and show the mapping existing between the data graph and the control graph.

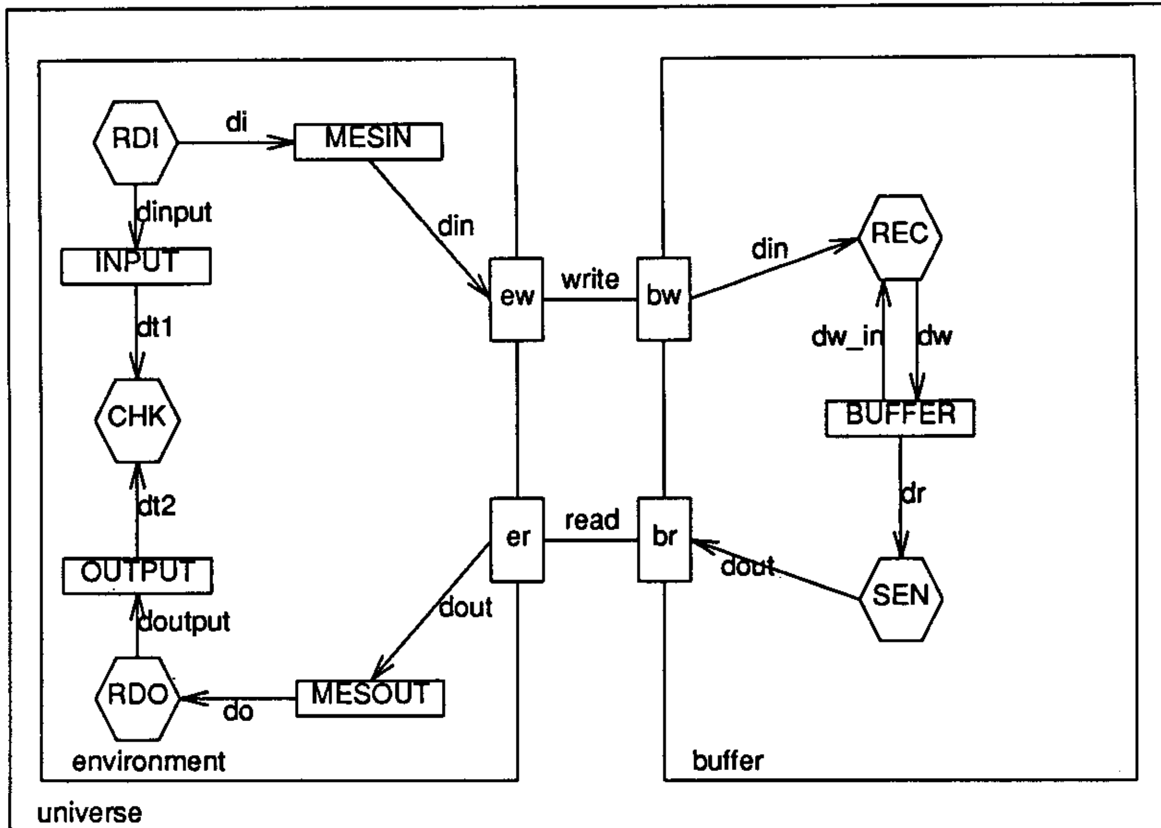


Figure 8.4: GMB Data Graph for the Buffer System

The following table describes the function of the various data processors and datasets in the data graph:

Controlled Processors

CHK	Mapped to control node TERM, processor which checks the message received against the initial messages to determine that they are the same and in the same order before termination.
RDI	Mapped to SEND in the control graph. It reads messages from INPUT into MESIN, one at a time.

RDO	Mapped to REC2 in the control graph. It reads messages from MESOUT, and deposits them into OUTPUT, one at a time.
REC	Mapped to RECM in the control graph. It receives messages from the environment and deposits them into dataset BUFFER.
SEN	Mapped to REQ in the control graph. It reads messages from BUFFER and passes them back to the environment.

Datasets

INPUT	Initial sequence of messages.
OUTPUT	Messages read from BUFFER, to be checked against INPUT.
MESIN, MESOUT	Message slots, interfaced with submodule buffer.
BUFFER	The actual buffer in the system.

8.4.3 The Interpretation Domain

The Interpretation Domain defines the format of the data stored in datasets and defines the transformations of data performed by the data processors. Many interpretation languages can be used for this domain. The original SARA system used PLIP (an extension of PL/1) as its interpretation language. The current system, being implemented in a Lisp dialect, supports a Lisp-like interpretation language.

8.5 Building Block Library

In order to support bottom-up design, it is necessary to have a collection of previously designed and tested models, appropriate for the design domain, stored in a design database. The SARA design database is called the Building Block Library by Drobman [Drob80]. His work concentrates on hardware building blocks but the procedure is also applicable to software modules.

The primary hypothesis of Drobman's work is that "a set of models of hardware and software building blocks can be created and utilized as primitive elements in a computer-aided design system and methodology such that the composition of requirements-satisfying, partially correct, microprocessor-based digital systems is dramatically enhanced." He demonstrated satisfaction of the hypothesis by defining building block descriptions of the Am2901 bit-sliced microprocessor, the Am29775 PROM, and other similarly complex devices, and then using those building blocks to design a 16-bit microprogrammable microprocessor.

Drobman's building blocks are prefabricated simulation models of physical building blocks. The simulation models are defined in the previously mentioned SARA languages, the Structure Model Language and the Graph Model of Behavior Languages.

Other SARA researchers have studied the requirements and organization of a design library [Land83, Land86, Mars83].

8.6 Socket Attribute Modeling

During research on the Building Block Library and the SARA simulation tools, it was felt that many of the errors detected during simulation could have been found much earlier by analysis of some as-of-yet undefined static description of the building blocks. This observation spawned SamPaio's research into the Socket Attribute Model, SAM [Samp79], and Penedo's research into the Module Interconnect Description, MID [Pene81]. While both deal with a description of a building block at its interfaces, sockets or interconnects, SAM concentrates on hardware building blocks and MID concentrates on software building blocks.

SamPaio provides a language to describe the behavioral attributes of a hardware module's sockets, for example, electrical characteristics (fan-in, fan-out), timing (set-up and hold times), bandwidth, and perhaps physical characteristics. With these descriptions attached to a module's sockets it is possible to detect inconsistencies ocurring during composition of two or more modules. The detection of socket mismatch errors ocurrs at the time the socket connection is attempted, not later during an expensive and time consuming simulation that may not detect the error at all.

8.7 Module Interconnect Description

Penedo attacks the same problem as SanPaio, but on the software front. She describes software modules as they appear at their interfaces. Most type checking compilers detect some of the errors that Penedo is after, for example, procedures called with the wrong number or type of arguments. The product of her research is the Module Interconnect Description, MID [Pene81, Pene79]. Berry later shows that Ada package specifications meet the needs of Penedo's MID [Berr84]. Krell [Krel86] continues this line of reasoning by researching the suitability of Ada as the language for the interpretation domain of the GMB.

8.8 Extensibility and User Interface

The extant SARA system implementation at MIT was not constructed in an ad hoc manner. From the beginning, the implementors knew that no matter how complete their tool kit was, there would be inevitable pressure to add new tools. They therefore established a procedure for constructing a new tool and for eventually integrating it with the existing tool kit. This procedure is described in [Vern78]. To insure consistency between existing and newly defined tools, Fenchel [Fenc80] defined a user interface construction tool that promotes sharing of grammatical con-

structs between tools. By following the procedure and by using the user interface construction tool, the end product is *self-describing* offering syntactic and semantic help to the end user. Fenchel's tool [Fenc82, Fenc78] is summarized in the following paragraphs.

Each tool intially is partitioned into user interface dependent and independent parts. The user interface independent part is partitioned into a collection of PL/1 routines that comprise the tool's functionality. The syntax of the user interface dependent part is described in an SLR(1) grammar. Upon recognition of certain syntax rules, a user interface independent routine is called.

Once the tool is fully constructed the user interface independent routines are merged with those of any pre-existing tools. The tool's syntax specification is added as a subdialogue to the tool system's grammar.

The underlying support tools use the grammar to provide integral help to the end user. This insures that the user gets help information that is in agreement with the implementation. It also alleviates the burden of providing help from the tool implementor.

8.9 The SARA/IDEAS Environment

The SARA methodology is supported by automated tools in an integrated, interactive environment. These tools allow the user to create and modify SM and GMB models. There is also a design data base from which models can be stored and retrieved.

The GMB Simulator [Razo79] is one of the tools in the SARA environment. Another tool is the Control Flow Analyzer, used to analyze control graphs for control

flow anomalies such as deadlocks and to assure some control flow properties such as proper termination.

All of these tools were developed using an early version of the method and supporting development software described in this dissertation. The SARA/IDEAS system runs on an APOLLO workstation at UCLA.

8.10 Summary and Conclusions

The SARA tools comprise a powerful, interactive, modeling environment. As such, the tool set is representative of CADOCS systems in existence today. The SARA/IDEAS system now being constructed incorporates all of the functionality of the previous SARA system. In addition, graphical interaction is incorporated. An even greater decoupling of interaction tasks and operational software, a more comprehensive and formal specification technique, and greater integration of design tools with the design data base, have all been achieved.

Specification and construction of SARA/IDEAS provides an excellent testbed for the method and for the tool-building tools put forth in this dissertation.

CHAPTER 9

Conclusions

As computer based systems become more complex and assume greater responsibilities, the need to produce reliable and cost-effective designs becomes increasingly important. Many tools have been proposed and built that support the design activity. When several of these tools are gathered together, the aggregation is called a Computer-Aided Design of Computer Systems, CADOCS, system.

Often, CADOCS systems are constructed from separately developed computer programs. Often these programs are written in different programming languages to run on different computer systems, offer different user interfaces, and store the results of their analysis or simulation in a way that precludes information sharing between programs. Tool developers typically concentrate their efforts on the design capability of a tool and are unable to justify the expense of providing a state-of-the-art user interface. The resulting CADOCS systems are only nominally integrated and are difficult to use at best. The power of such CADOCS systems is roughly equal to that provided by the set of comprising tools less that due to the difficulty of use caused by the juxtaposition of inconsistent user interfaces and of incompatible analysis outputs. The cost of CADOCS system construction is equal to the cost of developing the set of constituent tools plus the cost of masking over the inconsistencies and incompatibilities.

A major contribution of this research is the definition of a tool building system and a run-time environment that promotes the construction of a consistent and compatible set of tools whose increased utility derives from the synergistic aggregation of complementary tools and whose cost is lower due to the significant amount of software that is shared between tools.

The fundamental approach is to decouple interface software from modeling and analysis software. This allows implementation of each to proceed independently and to be implemented by specialists. The decoupling also allows several different interfaces to be proposed for a single implementation of modeling software.

The method and tools defined in this work represent a significant improvement in interactive system development. The capabilities of the previous SARA implementation have been extended to include the construction of graphical interfaces, access to a larger body of reusable code, and a more orderly evolution from tool conceptualization through to implementation. The work represents the synthesis of formal language theory, human-computer interaction, and software engineering applied to the specific problem of designing and implementing integrated collections of computer-aided design tools.

The SARA/IDEAS environment provides a fertile environment for exploration in the area of user interface. Experimentation will lead to a better understanding of what constitutes a good user interface. The support system also provides an umbrella under which additional CADOCS tools can be integrated with a system-wide, consistent user interface.

The methodology has successfully been applied to the definition of the SARA tools as well as to the definition of an Influence Diagram Editor which has

application in political science and policy analysis.

9.1 Contributions

User Interface Management System Kernel.

The methodical use of the tool building system produces a procedural interface to the modeling objects appearing in the user interface. The kernel provides another procedural interface to objects. All operations on objects are funneled through this deeply buried procedural interface consisting of do, undo, and redo procedures. The tool designer need not be aware of this interface. However, its services will be invoked automatically. A set of families of do, undo, and redo routines are defined along with a set of equations to help determine the space and time cost associated with the use of each family of routines.

Filtering operations through the funnel provides: necessary error recovery services for the design data base management system, design configuration management services, interactive user error recovery services, support for interactive exploration, audit trails, and a mechanism to support forward search and backtracking algorithms. A tool designer never has to justify the cost of developing bi-directional execution software; it is incorporated automatically.

A prototype implementation of the minimal state capture method of bi-directional execution defined in Chapter 7 is implemented in the T language under BSD 4.2.

Interaction Handler

The interaction handler, or ATN interpreter, is fully implemented on BSD 4.2 and on the APOLLO and fully integrated with SARA/IDEAS. Eduardo Krell translated a simple prototype written in Franz Lisp into the final product written in T. Eddie Lor and Krell replaced the ATN with LL(1) parse tables and rewrote the Interaction Handler accordingly.

The Interaction Handler couples the syntactic interface and the semantic modeling capabilities according to the syntax specification. Its services are shared by all tools in the SARA/IDEAS system. It provides command recognition, invocation of semantic routines, integral help, prompting, and menu generation.

Method.

The method is multilevel and object-oriented. The four phases of CADOCS tool design method are the conceptual or semantic, the grammatical, or syntactic, logical device, and physical device phases.

The semantic phase supports a designer in the creation of a new modeling capability from its initial conceptualization to its complete semantic definition. Modeling objects, their inter-relationships, and the operations performed upon them are all identified in this phase.

The grammatical phase supports complete syntactic definition of a tool. Primitive grammatical elements, terminals, are identified. Terminals can be ordered and collected into phrases and sentences, not-terminals. The tool designer will associate each sentence with a semantic operation defined in the previous phase. The run-time environment must support the recognition of sentences and an invocation of the specified operation.

The logical device phase focuses the designer's attention on the logical devices and logical interaction tasks required to support interaction. have been identified by other researchers. Software implementations of logical devices (e.g., selector, valuator, pick, text) are developed for available physical devices (e.g., a specific mouse, tablets, keyboards). These logical device drivers are maintained in a special library. This phase is primarily concerned with the mapping of terminals from the previous phase onto logical device drivers from the library.

Finally, the physical device phase focuses the designer's attention on the physical device inventory available to support interaction. Logical devices and tasks are assigned to physical devices. Several other aspects of the physical environment may also be dealt with in this phase.

Each phase of the method is supported by a specification language and compiler. The resulting set of specifications allow the tool designer four degrees of freedom in altering the final tool specification. For example, several alternative syntaxes can be proposed and implemented with little or no alteration of the other three phase's specification.

The Tool Building System Implementation

The parsers for the semantic phase and the physical device phase are implemented on BSD 4.2. They are not integrated with the other components of SARA/IDEAS on the APOLLO workstation.

The syntax phase compiler and the logical device phase compiler are implemented on BSD 4.2 and are fully integrated with SARA/IDEAS on the APOLLO. Eduardo Krell translated an LL(1) parse table generator written in Franz Lisp into the final syntax and logical device phase compilers written in T.

9.2 Suggestions for Further Research

As a result of this work, several areas for further research were identified. They are summarized in the following paragraphs.

Elevate Level of Interaction Tasks

The set of interaction tasks is too small and is defined at too low a level of abstraction. For example, moving a graphic object from one point on the display surface to another is a common operation in systems incorporating interactive graphics, yet describing a move operation using the interaction tasks discussed in this dissertation would have to be implemented, perhaps, as two picks. Another example is expanding a graphic object. Dragging and rubber-banding are common techniques that would be difficult to describe given this dissertation's set of interaction tasks. The set of interaction tasks and related logical devices should be extended with a set of higher level operations.

Relax Language Restrictions

Humans are used to conversing with computers by issuing simple imperative and interogative sentences. However, when several designers use a computer as a medium to support collaboration, imperative and interrogative sentence structure may prove to be too restrictive. Further research should be conducted to relax the LL(1) restriction without loss of prescience and the other ingredients necessary for intelligent, multi-level prompting, error detection, error reporting, error recovery, and user help. The Augmented Transition Network is capable of supporting far less restrictive grammars.

Apply Method to Tool-Building System

This work identifies an essentially batch means of specifying a system's user interface. Three steps are involved, specify the interface with an editor, compile it, execute it. Using this batch mechanism, it is possible to build an interactive tool that will replace the batch tools.

Extend Coverage to Include General Interactive Systems

The approach taken here to interactive modeling systems should be studied for its applicability to the broader class of general, interactive systems.

Rebuild System in Ada

The operation specification languages defined in this work evolved toward Ada package syntax as the research matured. Implementing the tool-building system and user interface management system in an Ada-like language may produce a more robust and elegant system.

Menu Language

The integral help mechanism is capable of showing the user all of the tokens that would be accepted by the interaction handler from the current state. This same information can be used to generate menus. The shape library could be augmented to include a display description of a system-wide menu.

Adding the capability of automatic menu generation would constitute a significant advance in the interfaces that can be described using the SARA tool-building system.

Multi-Party Dialogue

Designers must collaborate on complex designs. Most design systems prevent designers from interfering with each other, but no design system discovered during this research encouraged and supported design collaboration. The specification of multi-party dialogue appears to be a good starting point for research into design collaboration.

User Interface Experiments

This research discovered very little agreement on the meaning of a good interfaee. The SARA/IDEAS system collects all human-computer interaction in one place. It thus provides an excellent environment for conducting quantitative and qualitative user interface experiments.

REFERENCES

[Acqu82] Acquah, James, James Foley, and Patricia Wenner, ''Reference Manual for Advanced Raster Graphics Extensions to ACM/SIGGRAPH Core System,'' Tech. Rep. IIST Report 82-15, Department of Electrical Engineering and Computer Science, George Washington University, Washington, D.C., (March 1982).

[AED80] AED, *AED 512 Color Graphics Terminal: Control Protocol,* Advanced Electronics Design, Inc., Sunnyvale, California 94086 (May 1980).

[Aho72] Aho, A.V. and J.D. Ullman, *The Theory of Parsing, Translation, and Compiling: Volume I,* Prentice-Hall, Englewood Cliffs, New Jersey (1972).

[Aho73] Aho, A.V. and J.D. Ullman, *The Theory of Parsing, Translation, and Compiling: Volume II,* Prentice-Hall, Englewood Cliffs, New Jersey (1973).

[Alfo74] Alford, Mack W., I. Fennell Burns, Thomas Bell, Earl Benoit, and Jacob C. Richardson, ''Software Requirements Engineering Methodology Notebook,'' Tech. Rep. TRW-22944-6921-020, TRW Systems Group, Huntsville, Alabama, (31 October 1974).

[Alli77] Allison, James, *Structure Chart Graphics,* Hughes Aircraft Company (1977).

[Arch81a] Archer, James E. Jr. and Richard Conway, ''COPE: A Cooperative Programming Environment.,'' Tech. Rep. Tech. Rep. TR81-459, Department of Computer Science, Cornell University., Ithaca, N.Y., (June 1981).

[Arch81b] Archer, James E. Jr., *The Design and Implementation of a Cooperative Program Development Environment.,* Department of Computer Science, Cornell University., Ithaca, N.Y. (1981).

[Arch84] Archer, James E. Jr., Richard Conway, and Fred B. Schneider, ''User Recovery and Reversal in Interactive Systems,'' *ACM Transactions on Programming Languages and Systems* 6(1), pp. 1-19 (January 1984).

[Astr76] Astrahan, M. M., M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson, ''System R: A Relational Approach to Data Base Management,'' *ACM Transactions on Database Systems* **1**(2), pp. 97-137 (June 1976).

[Bake80] Baker, C. M. and C. Terman, ''Tools for Verifying Integrated Circuit Designs,'' *LAMBDA* (Fourth Quarter 1980).

[Balz83] Balzer, R., T.E. Cheatham, and C. Green, ''Software Technology in the 1990's: Using a New Paradigm,'' *IEEE Computer* **16**(11), p. 39 (November 1983).

[Barb79a] Barbacci, Mario, ''Instruction Set Processor Specifications for Simulation, Evaluation, and Synthesis,'' *Proceedings 16th Design Automation Conference*, pp. 64-72 (June 1979).

[Barb79b] Barbacci, Mario R., Gary E. Barnes, Roderic G. Cattell, and Daniel P.Siewiorek, *The ISPS Computer Description Language - Reference Manual*, Carnegie Mellon University (August 16, 1979).

[Barb80] Barbacci, Mario R., Andrew W. Nagle, and J. Duane Northcutt, *An ISPS Simulator*, Carnegie Mellon University (January 4, 1980).

[Beat82] Beatty, John C., ''On the Relationship Between LL(1) and LR(1) Grammars,'' *Journal of the Association for Computing Machinery* **29**(4), pp. 1007-1022 (October 1982).

[Beec85] " Beech, David, Christian Gram, Hans-Jürgen Kugler, Helmut Stiegler, and Claus Unger, *The IFIP WG 2.7 Reference Model for Command Response Languages*, The International Federation for Information Processing (September 1985).

[Berr83] Berry, D.M. and O. Berry, ''The Programmer-Client Interaction in Arriving at Program Specifications: Guidelines and Linguistic Requirements,'' *Proceedings of IFIP TC2 Working Conference on System Description Methodologies* (May, 1983).

[Berr84] Berry, Dan M., ''On the Use of ADA as a Module Interface Description,'' *Proceedings Hawaii International Conference on System Sciences* (January 1984).

[Bian76] Bianchi, M. H. and J. L. Wood, ''A User's Viewpoint on the Programmer's Work Bench,'' *Second International Conference on Software Engineering*, pp. 19-25, IEEE (October 1976).

[Bles82] Bleser, Teresa and James Foley, ''Towards Specifying and Evaluating the Human Factors of User-Computer Interfaces.,'' pp. 1-5 in *Proceedings of Human Factors in Computer Systems* (March 15-17, 1982).

[Boeh81] Boehm, Barry W., *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, New Jersey (1981).

[Bona77] Bonanni, L. E. and A. L. Glasser, *Source Code Control System User's Manual*, Bell Laboratories, Murray Hill, New Jersey (November 1977).

[Bonh76] Bonham, G. Matthew and Michael J. Shapiro, ''Explanation of the Unexpected: The Syrian Intervention in Jordan in 1970,'' pp. 113-141 in *Structure of Decision: The Cognitive Maps of Political Elites*, ed. Robert M. Axelrod, Institute of International Studies, University of California, Berkeley (1976).

[Brit81] Britton, Kathryn Heninger, R. Alan Parker, and David L. Parnas, "A Procedure for Designing Abstract Interfaces for Device Interface Modules," *Proceedings of the 5th International Software Engineering Conference* , pp. 195-204, IEEE Computer Society Press (March 1981).

[Brow83] Brown, Harold, Christopher Tong, and Gordon Foyster, "Palladio: An Exploratory Environment for Circuit Design," *IEEE Computer* **16**(12), pp. 41-56 (December 1983).

[Cai85] Cai, Shijie, Dorothy Landis, Dorab Patel, and Duane Worley, "An Experiment to Determine Appropriate Data Structuring Methods for the SARA-IDEAS Structure Model (SM)," Tech. Rep. INTERNAL MEMORANDUM #212, Computer Science Department, University of California, Los Angeles, (1985).

[Camp78] Campos, Ivan M. and Gerald Estrin, "SARA Aided Design of Software for Concurrent Systems," *Proceedings of the National Computer Conference*, Anaheim, California (June 1978).

[Card83] Card, Stuart K., Thomas P. Moran, and Allen Newell, *The Psychology of Human-Computer Interaction,* Lawrence Erlbaum Associates, Publishers, Hillsdale, New Jersey (1983). ISBN 0-89859-243-7.

[Chen76] Chen, Peter Pin-Shan, "The Entity-Relationship Model: Toward a Unified View of Data," *ACM Transactions on Database Systems* **1**(1), pp. 9-36 (March 1976).

[Chen83] Chen, Peter Pin-Shan, "English Sentence Structure and Entity-Relationship Diagrams," *Information Sciences* **29**, pp. 127-149 (1983).

[Chom63] Chomsky, N., "Formal Properties of Grammars," *Handbook of Mathematical Psychology* **2**, Wiley (1963).

[Ciam76] Ciampi, P. L., A. D. Donovan, and J. D. Nash, "Control and Integration of a CAD Data Base," pp. 394 -401 in *Proceedings 13th Design Automation Conference* (June 1976).

[Coyl81] Coyle, Geoffrey, "A Model of the Dynamics of the Third World War: An Exercise in Technology Transfer," *Journal of the Operational Research Society* **32**(9), pp. 755-765 (1981).

[Date77] Date, C.J., *An Introduction to Database Systems,* Addison-Wesley (1977).

[DoD80] DoD, Department of Defense, *STONEMAN: DoD Requirements for the Ada Programming Support Environment*, February 1980.

[Dolo76] Dolotta, T. A. and J. R. Mashey, "An Introduction to the Programmer's Work Bench," *Second International Conference on Software Engineering*, pp. 1-5, IEEE (October 1976).

[Drob80] Drobman, J. H., ''A Model-Based Design System and Methodology for Composition of Microprocessor-Based Digital Systems,'' Tech. Rep. PhD Dissertation, Computer Science Department, University of California, Los Angeles, (1980).

[Estr78] Estrin, Gerald, ''A Methodology for Design of Digital Systems - Supported by SARA at the Age of One,'' pp. 313-336 in *Proceedings of the National Computer Conference*, AFIPS (1978).

[Fair80] Fairley, Richard E., ''ADA Debugging and Testing Support Environments,'' *ACM* (1980).

[Feld82] Feldman, Michael and James Foley, ''Information Display Systems,'' Tech. Rep. Report, Department of Energy, (March 1982).

[Feld78] Feldman, S. I., *Make -- A Program for Maintaining Computer Programs*, Bell Laboratories, Murray Hill, New Jersey (August 1978).

[Fenc78] Fenchel, Robert S, ''SARA SLR(1) Grammar Tools,'' Tech. Rep. #186, UCLA Computer Science Department, Los Angeles, (November 1978).

[Fenc80] Fenchel, Robert S, ''Integral Help for Interactive Systems,'' Tech. Rep. UCLA-ENG-8051, UCLA Computer Science Department, Los Angeles, California, (September 1980).

[Fenc82] Fenchel, Robert S and Gerald Estrin, ''Self-Describing Systems Using Integral Help,'' *IEEE Transactions on Systems, Man and Cybernetics* **12**(2), pp. 162-167 (March-April 1982). special issue on User Assistance and Human Fators in Interactive Computer Systems.

[Fole81] Foley, James and Patricia Wenner, ''The George Washington University Core System Implementation,'' *Computer Graphics* **15**(3), pp. 123-132 (August 1981).

[Fole82a] Foley, J.D. and A. Van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley (1982).

[Fole82b] Foley, J.D. and A. Van Dam, ''Interaction Devices and Techniques,'' pp. 183-214 in *Fundamentals of Interactive Computer Graphics*, Addison-Wesley (1982).

[Forr61] Forrester, Jay W., *Industrial Dynamics*, The M.I.T. Press, Cambridge, Massachusetts (1961).

[Forr69] Forrester, Jay W., *Urban Dynamics*, 1969.

[Gaun82] Gaunpert, Gary and Robert Cathcart, *INTER/MEDIA*, Oxford University Press (1982). Interpersonal Communication in a Media World.

[Gold83] Goldberg, Adele and David Robson, *Smalltalk-80, The Language and its Implementation,* Addison-Wesley, Menlo Park, California (1983).

[Gold84] Goldberg, Adele, *Smalltalk-80, The Interactive Programming Environment,* Addison-Wesley, Menlo Park, California (1984).

[Habe79] Haberman, Nico, ''An Overview of the Gandalf Project,'' *CMU Computer Science Research Review* (1978-1979).

[Hamm75] Hammer, M. M. and D. J. Mcleod, ''Semantic Integrity in a Relation Data Base System,'' pp. 25-47 in *Proceedings on Very Large Data Base Conference*, Framingham, Mass. (September 1975).

[Hanl79] Hanlon, James, *Implementation and Evaluation of Input Functions for the 1979 CORE Graphics System,* George Washington University (1979).

[Hans73] Hansen, Per Brinch, ''A Case Study: RC 4000,'' in *Operating System Principles*, Prentice Hall, Inc., Englewood Cliffs, New Jersey (1973).

[Haye81a] Hayes, Mary Ann, ''Equations for ISDM Metrics,'' Tech. Rep. ISDM Software Engineering Notebook III:4:4, Hughes Aircraft Company, Fullerton, California, (20 July 1981).

[Haye81b] Hayes, Mary Ann, ''Concepts for ISDM Metrics,'' Tech. Rep. ISDM Software Engineering Notebook III:4:3, Hughes Aircraft Company, Fullerton, California, (21 May 1981).

[Held75a] Held, G. D., M. R. Stonebraker, and E. Wong, ''INGRES- A Relational Data Base System,'' in *Proceedings of the National Computer Conference* (1975).

[Held75b] Held, G. D. and M. R. Stonebraker, ''Storage Structures and Access Methods in the Relational Data Base Management System INGRES.,'' in *Proceedings ACM Pacific Conference*, San Francisco (April 1975). Available from Mail Room, Boole and Babbage Inc., 850 Stewart Drive, Sunnyvale, CA 94086.

[Hill79] Hill, D. and W. vanCleemput, ''SABLE: A Tool for Generating Structured, Multi-level Simulations,'' pp. 272-279 in *Proceedings of the 16th Design Automation Conference* (June 1979).

[Hugh80] Hughes Aircraft Company, ''A Palimpsest on Hughes Air Defense Systems,'' Tech. Rep. ID 80-17-22, NATO Programs Department, Air Defense/Control Systems Laboratory, Software Engineering Division, Ground Systems Group, Hughes Aircraft Company, (1980).

[IBM73] IBM, ''OS/VS2-SYSTEMS HASP II, Logic,'' Tech. Rep. GY27-7255-0, (March 1973).

[IBM75] IBM, "OS/VS2-SYSTEMS JES2, Logic," Tech. Rep. SY28-0622-2, (April 1975).

[ISO81] ISO, International Standards Organization, *Graphical Kernel System (GKS) version 6.6*, May 1981.

[Jack81] Jackson, Mel, "Report on the Study of an ADA Based System Development Methodology," Tech. Rep. ID unknown, British Department of Industry, London, England, (September 1981).

[Jack83] Jackson, Michael A., *System Development,* Prentice Hall International, Englewood Cliffs, New Jersey (1983).

[Jaco83] Jacob, Robert J. K., "Using Formal Specifications in the Design of a Human-Computer Interface," *Communications of the ACM* (1983). Naval Research Laboratory.

[Jens83] Jensen, Edward, "Automated Interactive Design and Evaluation System," Tech. Rep. 1851132-1, Hughes Aircraft Company, (1 November 1983). Volume VI - Primer.

[Joha78] Johansen, Robert and Robert DeGrasse Jr., and Thaddeus Wilson, "Group Communication Through Computers," Tech. Rep. R-41, Institute for the Future, Menlo Park, California, (February 1978). Effects on Working Patterns.

[John75] Johnson, S.C., "YACC - Yet Another Compiler-Compiler," Tech. Rep. Computer Science Technical Report No. 32, Bell Laboratories, (July 1975).

[John78] Johnson, S.C. and M.E. Lesk, "Language Development Tools," *Bell Systems Technical Journal* 57(6,2), pp. 2155-2176 (July-August 1978).

[Jone81] Jones, Robert R., "Distributed Data Processing Modeling for Future ATC Systems," in *Proceedings of 25th Air Traffic Control Association,* Arlington, Virginia (October 1981).

[Joy80a] Joy, William, *An Introduction to Display Editing with Vi,* Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley (September 16, 1980).

[Joy80b] Joy, William, *An introduction to the C shell,* Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley (November 8, 1980).

[Katz83] Katz, Randy H., Joseph Moran, Umakishore Ramachandran, and Daniel Schuh, "Big Caesar and Little Caesar: Adventures in Modifying and Extending CAD Software," *VLSI Design*, pp. 58-60 (January/February 1983).

[Kern77] Kernighan, Brian W., ''PIC - A Graphics Language for Typesetting,'' Tech. Rep. Bell Laboratories Computing Science Technical Report xx, (1977).

[Knut65] Knuth, Donald E., ''On the Translation of Languages from Left to Right,'' *Information and Control* 8(6), pp. 607-639 (December 1965).

[Knut68] Knuth, Donald E., ''Semantics of Context-Free Languages,'' *Math. Systems Theory* 2(2), pp. 127-145 (June 1968).

[Kras83] Krasner, Glenn editor, *Smalltalk-80, Bits of History, Words of Advice,* Addison-Wesley, Menlo Park, California (1983).

[Krel86] Krell, Eduardo Aaron, ''ADA as an Interpretation Language for SARA,'' Tech. Rep. PhD Dissertation, Computer Science Department, University of California, Los Angeles, (1986).

[Land83] Landis, Dorothy, ''Design Considerations for the Satisfaction of CAD Library Requirements,'' Tech. Rep. UCLA CSD-830614, (June 1983).

[Land86] Landis, Dorothy, *CADIS: A Kernel Approach Toward the Development of Intelligent Data Management Support for Computer Aided Design Systems,* UCLA Computer Science Department (June 1986).

[Lewi76a] Lewis, Philip M. II, Daniel J. Rosenkrantz, and Richard E. Stearns, *Compiler Design Theory,* Addison-Wesley (May 1976).

[Lewi76b] Lewis, Philip M. II, Daniel J. Rosenkrantz, and Richard E. Stearns, ''Top-Down Processing,'' pp. 227-288 in *Compiler Design Theory,* Addison-Wesley (May 1976).

[Lill81] Lillehagen, Frank M., ''CAD/CAM Work Stations for Man-Model Communication,'' *IEEE Computer Graphics and Applications* 1(3), pp. 17-27 (July 1981).

[Mars83] Marshall, Joan, ''A Design Automation Database for Computer Aided Design of Computer Systems,'' *Phd Dissertation*, Computer Science Department, University of California, Los Angeles (1983).

[Mash76a] Mashey, J. R., ''Using a Command Language as a High-Level Programming Language,'' *Second International Conference on Software Engineering*, pp. 6-13, IEEE (October 1976).

[Mash76b] Mashey, J. R. and D. W. Smith, ''Documentation Tools and Techniques,'' *Second International Conference on Software Engineering*, pp. 14-18, IEEE (October 1976).

[McDo74] McDonald, N., M. R. Stonebraker, and E. Wong, ''Preliminary Design of INGRES. Part 1: Query Language, Data Storage and Access.,'' Tech. Rep. Electronics Research Laboratory Memorandum ERL-M435, Berkeley: University of California., (April 1974).

[Mead80] Mead, Carver and Lynn Mead, *Introduction to VLSI Systems,* Addison-Wesley (1980).

[Mill79] Miller, Lawrence H., ''Experimental Validation of Cognitive Models for Man-Machine Interaction,'' *Teleinformatics, '79,* Springer-Verlag (June, 1979).

[Newe72] Newell, A. and H. Simon, *Human Problem Solving,* Prentice Hall, Englewood Cliffs, N.J. (1972).

[Oust81] Ousterhout, J. K., ''Caesar: An Interactive Editor for VLSI Layout,'' *VLSI Design,* pp. 34-37 (Fourth Quarter 1981).

[Parn79] Parnas, David L., ''Designing Software for Ease of Extension and Contraction,'' *IEEE Transactions on Software Engineering* SE-5(2), pp. 128-137 (March 1979).

[Pene79] Penedo, Maria H. and Dan M. Berry, ''The Use of a Module Interconnection Language in the SARA System Design Methodology,'' *Proceedings of the 4th International Conference on Software Engineering* (September 1979).

[Pene81] Penedo, Maria H., ''The Use of a Module Interface Descripton in the Synthesis of Reliable Software Systems,'' Tech. Rep. PhD Dissertation, Computer Science Department, University of California, Los Angeles, (1981).

[Pete81] Peterson, J. L., *Petri Net Theory and the Modeling of Systems,* 1981.

[Pete83a] Peterson, James Lyle and Avi Silberschatz, ''Design Principles,'' in *Operating System Concepts,* Addison-Wesley Publishing Company, Inc. (1983).

[Pete83b] Peterson, James Lyle and Avi Silberschatz, ''Design Principles,'' pp. 422 in *Operating System Concepts,* Addison-Wesley Publishing Company, Inc. (1983).

[Razo77] Razouk, Rami R. and Gerald Estrin, ''The Graph Model of Behavior,'' *Proceedings of the Symposium on Design Automation and Microprocessors,* pp. 67-76, IEEE Piscataway, New Jersey (December 1980).

[Razo79] Razouk, Rami R., Mary Vernon, and Gerald Estrin, ''Evaluation Methods in SARA -- The Graph Model Simulator,'' *Proceedings of Conference on Simulation, Measures and Modeling of Computer Systems,* pp. 189-206 (1979).

[Reme71] Remer, F.L. De, ''Simple LR(k) grammars,'' *CACM* **14**(7), pp. 453-460 (July 1971).

[Samp79] Sampaio, A.B.C., ''A Scheme of Attributes to Detect Inconsistencies in Design of Computer Systems,'' Tech. Rep. PhD Dissertation Prospectus, Computer Science Department, University of California, Los Angeles, (1979).

[Solo84] Soloway, Elliot, ''A Cognitively-Based Methodology for Designing Languages / Environments / Methodologies,'' pp. 193-196 in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, Penn. (May 1984).

[Stic77] Stickney, Mike E., *An Application of Graph Theory to Software Test Data Selection,* Hughes Aircraft Company (1977).

[Ston76] Stonebraker, M. R., ''Getting Started in INGRES-A Tutorial,'' *ACM Transactions on Database Systems* **1**(3), pp. 189-222, Berkeley: University of California. (August 1976).

[Sun 84] Microsystems, Sun Inc., ''Programmer's Reference Manual for SunWindows, the Sun Window System,'' Tech. Rep. Part Number 800-1116-01, Mountain View, California, 94043, (January 7, 1984).

[Tich82a] Tichy, W.F., ''Design Implementation, and Evaluation of a Revision Control System,'' *IEEE*, pp. 58-67 (1982).

[Tich82b] Tichy, W.F., ''A Data Model for Programming Support Environments and its Application,'' *Automated Tools for Information Systems Design-IFIP*, pp. 31-47, North-Holland Publishing Company (1982).

[Tiec74] Tiechroew, D., E. Hershey, and M. Bastarche, ''An Introduction to PSL/PSA,'' *ISODS Working Paper*(86), Department of Industrial and Operations Engineering University of Michigan (March 1974).

[Vall75] Valle, Jacques, Robert Johansen, Hubert Lipinski, Kathleen Spangler, Thaddeus Wilson, and Andrew Hardy, ''Group Communication Through Computers,'' Tech. Rep. R-35, Institute for the Future, Menlo Park, California, (October 1975). Pragmatics and Dynamics.

[Vall77] Valle, Jacques, Kathleen Spangler, and R. Garry Shirts, ''The Camelia Report,'' Tech. Rep. R-37, Institute for the Future, Menlo Park, California, (February 1977). A Study of Technical Alternatives and Social Choices in Teleconferencing.

[Vern78] Vernon, Mary, Robert Fenchel, and Rami Razouk, ''Standard Procedure for Designing a New Tool for the SARA System,'' Tech. Rep. Internal Memorandum 185, Computer Science Department, University of California, Los Angeles, (November 1978).

[Wall81] Wallin, Joe, *Integrated Software Design Methodology,* Hughes Aircraft Company (1981).

[Wass82] Wasserman, Anthony I., ''The User Software Engineering Methodology: an Overview,'' Tech. Rep. Technical Report #56, Medical Information Science, University of California, San Francisco, (July 1982).

[Wass84a] Wasserman, Anthony I., ''Characteristics of the User Software Engineering Methodology,'' in *Proceedings of the Software Process Workshop*, Medical Information Science, University of California, San Francisco (February 1984).

[Wass84b] Wasserman, Anthony I., ''Developing Interactive Information Systems with the User Software Engineering Methodology,'' in *Proceedings, First IFIP Conference on Human-Computer Interaction*, Medical Information Science, University of California, San Francisco, London, England (September 1984).

[Weav68] Weaver, Richard M., *A Concise Handbook*, Holt, Rinehart and Winston (1968).

[Weth81] Wetherell, C. and A. Shannon, ''LR - Automatic Parser Generator and LR(1) Parser,'' *IEEE Transactions on Software Engineering* SE-7(3), pp. 274-278, Lawerence Livermore Laboratory (May 1981).

[Will77] Willis, Ron R., *DAS - An Automated System to Support Design Analysis*, Hughes Aircraft Company (1977).

[Will78] Willis, Ron R., *Design Evaluation of Distributed Data Bases and Data Base Management Using Hughes' Distributed Data Processing Simulation Model (DDPM)*, Hughes Aircraft Company (1978).

[Winc81] Winchester, James W., ''Requirements Definition and its Interface to the SARA Design Methodology Based Systems,'' Tech. Rep. PhD Dissertation, Computer Science Department, University of California, Los Angeles, (1981).

[Wood70] Woods, W. A., ''Transition Network Grammar for Natural Language Analysis,'' *Communications of the ACM* 13(10), pp. 591-606 (October 1970).

[Worl80] Worley, Duane R., *Towards Software Reliability Through Testing*, Hughes Aircraft Company (1980).

[Worl85] Worley, D. Robert, *A Proposal for Decision Aids for the Rand Strategy Assessment Center*, The Rand Corporation (1985).

[Your79] Yourdon, Edward and Larry L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program Design.*, Prentice-Hall, Englewood Cliffs, New Jersey (1979).