INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700  800/521-0500
NOTE TO USERS

The original manuscript received by UMI contains pages with slanted print. Pages were microfilmed as received.

This reproduction is the best copy available

UMI
A specification paradigm for
design and implementation of non-WIMP
human-computer interactions

A dissertation

submitted by

Stephen A. Morrison

In partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in

Computer Science

TUFTS UNIVERSITY

May 1998

© 1998, Stephen A. Morrison

Adviser:

Dr. Robert J. K. Jacob
ABSTRACT

Most current user interface specification languages and toolkits are based on serial, discrete, token exchange paradigms which, in general, perform an acceptable job of implementing traditional WIMP (Window, Icon, Menu, Pointer) interfaces. These tools, however, are ill suited to address the needs of emerging interaction styles such as virtual environments. These interaction styles commonly rely upon: full duplex, asynchronous, interrelated dialogues; a blend of continuous and discrete inputs and responses; and, implicit commands and probabilistic input events. Some forms of non-WIMP interactions must also contend with real time processing constraints and deadline-based computations. This work proposes a specification paradigm, the SHADOW System, which provides a framework of techniques and abstractions which directly addresses these issues. This system has been demonstrated to allow both the semantic meaning and behavior of all interface elements to be clearly defined in a reusable fashion while providing support for software engineering practices such as modular design, complexity management, extensibility and traceability. Additionally, the system addresses issues of performance, portability, and maintainability, providing mechanisms which allow run time performance criteria and deadline contingency plans to be specified in a manner which is easily discernible and platform independent.

The SHADOW System provides a graphical specification language consisting of data flow graphs and augmented transition networks. This language is highly declarative in nature, supports loosely coupled, modular design and relies upon a run time engine to resolve constraints and to manage conceptually parallel tasks within uni-processing environments. The run time engine provides the infrastructure which allows the interface designer to address conceptual and semantic issues of
design without becoming entangled in the details of the implementation. Additionally, the engine provides facilities which automatically adjust processing loads to meet designer specified performance criteria should CPU processing time consumption become a problem.

This work details the proposed system as well as providing examples and analysis of several applications developed within the specification paradigm, providing a guide to the workings, features and limitations of the SHADOW System and its ability to meet the emerging needs of interface designers.
# TABLE OF CONTENTS

## SECTION I: Introduction
- Background 2
- Interaction Paradigms 4
- User Interface Management Systems 7
- The Specification Problem 10
- The SHADOW System 14

## SECTION II: Related Work
- Interface Design Concepts 15
- Architectural Considerations 17
- Specification Languages 20

## SECTION III: Problem Refinement
- Overview 24
- Conceptual Continuity Issues 24
- Real Time Behaviour Issues 26
- Scaleability Issues 30
- Extensibility Issues 32
- Modularity Issues 36
- Reusability Issues 38
- Intellectual Investment Issues 41
- Closing Comments 41
## SECTION IV: The SHADOW Talk Model

- Overview 42
- Subsystem Structure 45
- Constraint Management 46
- Link Elements 48
- Variable Elements 55
- Data Flow Network Specification 60
- Event Management 63
- Transition Network Specification 66
- Generic Constraint Management 69
- Constraint Scoping and Property Binding 70
- Generic Constraint Specification 74

## SECTION V: The SHADOW Run-Time Environment

- Overview 77
- General Processing Paradigm 79
- The Run Time Executive Manager 84
- The Automata Maintenance Facility 88
- The Constraint Solver Facility 93
- The Output Generation Facility 100
- The Polled Link Evaluation Facility 102
- The Generic Constraint Management Facility 105
- Dynamic Load Management 107

Table of Contents
SECTION VI: User Interface Development Under

The SHADOW System

Overview 115

Sequential Dependencies And Parallel Processing 119

Event Abstraction And Propagation 125

Data Abstraction 131

Functional Decomposition 136

Planning For Decimation 138

Virtual Subsystem Management 146

Generic Constraint Subsystems 150

Specifying Link Internals 153

Device Management 155

Closing Comments 159

SECTION VII: Evaluating The Shadow Paradigm 161

Overview 161

A Trivial Application 162

Development Of A Widget Set 165

Development Of A Text Editor 179

Programming Constraint Driven Behavior 188

Development Of A Gesture Recognition Application 203

Closing Comments 213
SECTION VIII: Design of a Virtual World

Overview 215
Decomposing The World 216
Establishing The Setting 218
Defining The Basic Actors 227
Creating A Complex, Self-Motivated Actor 236
Defining Ensemble Troupes 249
Directing A Cast Of Virtual Actors 258
Device Control And Modelling The User 260
Closing Comments 272

SECTION IX: Results

Overview 274
Addressing Conceptual Continuity 274
Addressing And Evaluating Real Time Behavior 277
Addressing Scaling Issues 282
Addressing Extensibility 284
Supporting Modular Design 285
Addressing Code Reuse 286
Preserving Intellectual Investments 287
Assessment Of The Expressive Power Of Shadow Talk 288

SECTION X: Conclusions 294

Table of Contents
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPENDIX A</td>
<td>296</td>
</tr>
<tr>
<td>APPENDIX B</td>
<td>320</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>326</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

1-1:  Contrasts in the underlying assumption between WIMP and non-WIMP interfaces 9
2-1:  Implementation strata as proposed by Foley and van Dam 17
4-1:  The three major components of a SHADOW subsystem specification 45
4-2:  The matrix of link classes and their graphical representations 51
4-3:  The graphical interface specification of a link element 52
4-4:  Alternate icons for a link element which returns the minimum of four input values 53
4-5:  The available variable classes and their graphical representations 56
4-6:  The available variable data types and their intended purposes 59
4-7:  Connectivity visualization conventions for data flow 61
4-8:  A plugboard specification of a simple slide bar 62
4-9:  The available variable classes and their graphical representations 67
4-10: Generic constraint property binding and association specification 71
4-11: Specification of a two object generic constraint property binding diagram with filter function 74
4-12: Simple gravity constraint data flow specification 76
5-1:  The architecture of a SHADOW based application 83
5-2:  The architecture of the SHADOW run time engine 85
5-3:  The interrelationships of the automata maintenance entities 89
5-4:  Abstractions of data flow graph elements 94

Table of Figures
5-5: A master data flow graph and its potential subgraphs
5-6: The evaluation process of the constraint solver facility
5-7: Overview of the output generation facility
5-8: Overview of the polled link evaluation facility
5-9: Partial overview of the generic constraint management facility
5-10: A data flow graph with a strong potential for redundant processing
6-1: The code translation process
6-2: Three possible evaluation orders for a dependency graph
6-3: A trivial example of a system requiring a particular sequence of evaluation
6-4: A simple application of a dependency thread
6-5: The effective scope resulting from various event distribution methods
6-6: Implementing manual decimation within the SHADOW paradigm
6-7: A simple pruning example
6-8: A simple replacement example
6-9: The effect of run levels on replacement decimation
6-10: An example of virtual subsystem use
6-11: Generic constraint scoping within the containment hierarchy
7-1: The SHADOW Talk answer to "Hello, World!"
7-2: A specification for a push button widget
7-3: The anatomy of a proportional slider widget
7-4: A specification for a horizontal slider with a proportionally sized handle
7-5: The composition of buttons and a slider to form a scrollbar widget
7-6: A specification for a horizontal scrollbar with a proportionally sized handle
7-7: A specification for scrolled screen region

Table of Figures
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-8</td>
<td>A simple text editor built from SHADOW WIMP widgets</td>
<td>180</td>
</tr>
<tr>
<td>7-9</td>
<td>The plugboard specification for the Notepad application</td>
<td>182</td>
</tr>
<tr>
<td>7-10</td>
<td>The augmented transition network specification for the Notepad application</td>
<td>183</td>
</tr>
<tr>
<td>7-11</td>
<td>Screenshot of the Notepad application showing menu and dialog widgets</td>
<td>185</td>
</tr>
<tr>
<td>7-12</td>
<td>The popup dialog box manager for the Notepad application</td>
<td>187</td>
</tr>
<tr>
<td>7-13</td>
<td>Screen layout of the Blocks application</td>
<td>189</td>
</tr>
<tr>
<td>7-14</td>
<td>Elemental decomposition of the Blocks program</td>
<td>190</td>
</tr>
<tr>
<td>7-15</td>
<td>The system level specification for the Blocks application</td>
<td>191</td>
</tr>
<tr>
<td>7-16</td>
<td>The screen layout specification for the Blocks application</td>
<td>192</td>
</tr>
<tr>
<td>7-17</td>
<td>The specification of a bounding wall in the Blocks application</td>
<td>193</td>
</tr>
<tr>
<td>7-18</td>
<td>Specification of the Blocks application's Paddle</td>
<td>195</td>
</tr>
<tr>
<td>7-19</td>
<td>The Ball subsystem of the Blocks application</td>
<td>196</td>
</tr>
<tr>
<td>7-20</td>
<td>Specifying an array of bricks using virtual subsystems</td>
<td>197</td>
</tr>
<tr>
<td>7-21</td>
<td>A brick specification which uses an external application to self-initialize</td>
<td>199</td>
</tr>
<tr>
<td>7-22</td>
<td>A generic constraint implementing linear motion</td>
<td>200</td>
</tr>
<tr>
<td>7-23</td>
<td>A generic constraint to detect and service two body collisions</td>
<td>202</td>
</tr>
<tr>
<td>7-24</td>
<td>The desktop gesture recognition program in operation</td>
<td>205</td>
</tr>
<tr>
<td>7-25</td>
<td>The stroke components of the three recognized gestures</td>
<td>206</td>
</tr>
<tr>
<td>7-26</td>
<td>The system level specification for the 2D Gesture application</td>
<td>208</td>
</tr>
<tr>
<td>7-27</td>
<td>The plugboard specification for recognizing the copy gesture</td>
<td>209</td>
</tr>
<tr>
<td>7-28</td>
<td>The augmented transition network for recognizing the copy gesture</td>
<td>212</td>
</tr>
<tr>
<td>8-1</td>
<td>A scene from the Rookery application</td>
<td>217</td>
</tr>
<tr>
<td>8-2</td>
<td>A top view of the scene layout for the Rookery</td>
<td>220</td>
</tr>
<tr>
<td>8-3</td>
<td>The specification of the basic scene</td>
<td>221</td>
</tr>
</tbody>
</table>
8-4: Specification of a two pan balance 223
8-5: The TimeIndex utility subsystem 225
8-6: The physics manager for the Rookery application 226
8-7: A generic subsystem which enables its container to be picked up and carried 229
8-8: A generic subsystem to map the characteristics of the user's hand to an object 231
8-9: The specification of a Ball subsystem 232
8-10: The specification of a Weight subsystem 234
8-11: The specification of a Teapot subsystem 235
8-12: The articulation capabilities of the penguin graphical object 236
8-13: The plugboard specification for the Penguin Subsystem 239
8-14: The augmented transition network for the Penguin subsystem 240
8-15: A subsystem to manifest a penguin's fascination 243
8-16: A subsystem to control the activity of an idle penguin 244
8-17: A subsystem to manage the panic reaction of a penguin 246
8-18: A subsystem to animate the waddling motion of a penguin 248
8-19: A subsystem designed to encourage a penguin to chase a ball 251
8-20: Coupling three penguins to play follow-the-leader 252
8-21: Specification of the user's penguin fan club 253
8-22: Specification of the tea chaser troupe 256
8-23: A pair of tea chaser penguins and their quarry 257
8-24: The initializing and managing the cast of the the Rookery 259
8-25: The main system definition for the desktop version of the Rookery 261
8-26: The desktop I/O controller module 263
8-27: The VRPort device Performer driver interface 265

Table of Figures
8-28: The MouseHand controller module 266
8-29: The Hand subsystem 267
8-30: The main system definition for the head-mounted version of the Rookery 269
8-31: The HeadMountIO module 270
8-32: The IScan device interface module 271
9-1: The run time behavior of the decimation test program 279
9-2: Overall performance resulting from automated decimation 280
9-3: C++ utilization in the design of the Rookery 290
9-4: Complexity measure for the Rookery 291
A specification paradigm for
design and implementation of non-WIMP
human-computer interactions
SECTION I: INTRODUCTION

BACKGROUND

In the past thirty years, the paradigms for human-computer interaction (HCI) and the popular demand for these evolving styles have changed as dramatically as the computer industry itself. By the time industry, government and academia had adopted scripted batch processing as the de facto standard interaction technique, the advent of timesharing had already given rise to command line dialogs. As computers moved farther from the labs of highly trained personnel and into the hands the (frequently deadline driven) public, the deficiencies of the command line interactions became apparent (steep learning curve requiring extensive memorization of both syntactic and semantic details). Alternate styles of interaction evolved to address these issues including question and answer dialogs, menus, and query forms [125].

The mass audience (at the time dominated by an eighty column, text only window on the computer's world) had no sooner come to accept (and indeed expect) these interaction styles when advances in the computer hardware industry made graphical displays economically possible. This innovation, combined with the rapidly expanding profile of a 'typical' end user, eventually lead to the development of 'Window, Icon, Menu, Pointer' (WIMP) style interfaces [41]. This is the basic theme around which many of the most popular graphical user interfaces in use today have been built.
History, however, has shown that users' expectations and preferences are subject to frequent change suggesting that, as more resources become available, the general public will become increasingly less satisfied with the current 'best' solution to the interaction problem. This is not to say that current WIMP style interfaces will become extinct, indeed job control languages and command shells are still actively being used by a significant portion of the user community despite the existence of (what are in theory) more user friendly alternatives, only that there will be an increasing number of applications of computer technology for which WIMP based interaction will be less than optimal in the eyes of the user's expectations. As a trivial example, entering a desired altitude into a field and clicking on a Bank Left button may be a perfectly acceptable way of controlling a flight simulator but the average game enthusiast would much rather use a joystick resembling a control yoke, and more to the point, the user's knowledge of the joystick as a valid input mechanism reduces the former solution to the realm of unacceptability.

Rising on the heels of traditional Graphical User Interface (GUI) systems (and fed to the public's expectations via Hollywood's often over-sensationalized presentation of virtual reality systems) non-WIMP style interfaces are of growing interest. Non-WIMP interactions often go beyond the simple event driven, point and click style behavior and allow a more continuous conceptual paradigm where the user directly manipulates objects within the computer's virtual environment in a manner consistent with the user's real world based expectations.

This chain of events leads to the conclusion that as user interaction styles continue to evolve, so must the tools and development paradigms which are used to specify and implement them [101]. This report discusses the issues and concerns that must be considered to effect such a
step in the evolution of HCI design tools and proposes a visual user interface description language (UIDL), the SHADOW System, which addresses many of these items.

**INTERACTION PARADIGMS**

The WIMP interaction paradigm is inherently a serial, event driven dialog where signals from a limited number of input devices are entered into a single input event queue. Events are handled in the order received and, as feedback becomes available, system responses are submitted to the output channel(s). Additionally, the absence of input events usually translates into an idle state for the system as a whole and all processing other than polling for input is suspended until new stimuli arrive.

The discretized single input event queue both hides and enforces certain assumptions about the manner in which humans are 'intended' to interact with the machine under the WIMP paradigm. First, all input channels are seen to be inherently separable and concurrent actions on the user's part are tokenized by the system as individual, unrelated events. The input queue is then filled on a first come first served basis from the various input channels and passed along to the application for processing. While this assumption is often safe if not preferable in the design of office automation software (dragging the mouse while typing is rarely a practical command token), it becomes a hinderance should a system designer desire to structure a user interface in which multiple input channels operate (at least conceptually) in parallel and in which tandem input signals are given semantic meaning.
A second assumption typical of WIMP systems is that, from a programmer standpoint, there is no such thing as a continuous operation even in the cases where the user’s perspective suggests a continuous action is intended. For example, a mouse drag operation is conceptually a continuous operation bracketed by two discrete events. From the user’s perspective, a mouse button was depressed (discrete action) and the location of the dragged object became a direct function the pointer location (continuous relationship) until the mouse button was released (discrete action). From a programmer standpoint, the entire gesture was modelled as a stream of mouse motion events (one for every pixel traversed), again bracketed by the button actions. While this assumption does not limit or hinder the processing of input signals (indeed, any input signal must be discretized at some level in order to be processed by a digital computer), it is unfortunate from a design and maintenance standpoint that the details of the implementation are not analogous to the intended conceptual relationships and actions.

A third assumption underlying the WIMP paradigm is that the sequence of events is the only timing issue of innate importance. The relative time between events as well as the time required to process sequentially the events in queue are, in general ignored. In many cases, delays in response time are considered an acceptable trade off in favor of the accurate, detailed responses eventually being produced. The impact of this assumption is mitigated by the half-duplex interaction used by WIMP systems; as the system becomes bogged down in processing and ceases to give feedback, a typical user will pause before generating additional input tokens indirectly giving the machine more time to catch up. Again, for most current applications, particularly in the area of office automation and data processing, this alternating turn interaction is considered acceptable despite its inherent risk of resource (both machine and human) underutilization and ignorance of real time deadlines.
Many emerging interaction styles and input/output devices fall outside the domain of tokens and responses that can be readily modelled using the WIMP paradigm. Such "non-WIMP" user interfaces seek to provide "non-command", parallel continuous, multi-mode interaction. Common examples of such interactions may be found in virtual reality systems and interactive simulations but are not limited to direct manipulation environments. The fundamental concepts of non-WIMP interactions may also be seen in handicap access interfaces, gesture-based interfaces, pen-based interfaces, eye-tracking based interfaces, intelligent agent interfaces, embedded system device interfaces, ubiquitous computing and state of the art computer games.

Non-WIMP interfaces have the potential to capitalize on the users' familiarity with real world interactions [126]. Rather than requiring the end user of an application to adapt to the application's designers' opinion as to how a particular task should be performed, non-WIMP systems draw upon the user's 'natural' reaction to a problem statement and associate the systems responses accordingly. For example, a user faced with the task of relocating a virtual object may grab the object, pick it up, move it and release it in the intended location. If it is inconvenient to carry the object all the way to it destination, the user might, in his or her virtual world, elect to throw the object in the hopes of it landing in the appropriate spot.

In theory this capturing of natural behavior makes user interfaces easier to learn and operate. In practice, however, the design and implementation of such systems is a difficult task. This is in part due to psychology and the mind set of the user community as a collective, and in part due to the lack of proper design and development tools to address the needs of the designers. The former problem is simply the latest incarnation of an issue which has been inherent in the
field of human-computer interaction since its inception. The 'natural' reaction for one person may vary greatly from one person to the next depending on the individual's life experiences. A person faced with the task of communicating with another at a remote location has several options available in the real world such as e-mail, phone, fax, hand written letter, and a very loud shout. Which technology would be deemed the 'natural' way for humans to communicate in the given situation is a function of the mind set of the users in question. While this problem is not one that can be easily solved, it does highlight the importance of the latter problem, that being the existence of robust tools and techniques to allow such interfaces to be designed and built in an efficient, reuseable, and maintainable fashion.

**USER INTERFACE MANAGEMENT SYSTEMS**

The need for tools to assist in the design and implementation of user interfaces is by no means limited to the scope of non-WIMP interaction paradigms. A broad class of tools, User Interface Management Systems (UIMS), has been in existence for some time to address this need [108]. There exist a multitude of UIMS software packages, each customized and optimized to address the needs and requirements of particular interaction styles.

In general terms, a UIMS is a collection of software development tools used to design, prototype, execute and evaluate user interfaces [87]. Included among its various components are usually some form of user interface description language (UIDL) and user interface execution engine. The UIDL allows the program designer to specify the nature and scope of human-computer interaction in high level terms, detached from the technical details of the actual
implementation. The execution engine manages and controls the dialog between the user and the underlying application in accordance with the directives given by the specification.

The use of a UIDL to specify the overall behavior of a system affords the system designer several benefits. First and foremost, the UIDL provides a layer of conceptual abstraction where the maturity and usability of the design may be discussed, evaluated and refined in a manner independent of its physical implementation. Additionally, once an interface has been specified using a UIDL, a corresponding user interface management system may be used to interpret or compile the specification itself directly into a machine executable format, eliminating the need for manual, low level code generation for user interface modules. From a software engineering standpoint, the generated code itself is disposable while the underlying libraries of routines (or toolbox) used by that code is fully reusable. The burden of system maintenance and evolution is passed on to the level of maintaining the user interface description rather than the implementation. The maintenance of the UIMS itself must also be considered but this cost can usually be amortized in that the UIMS is (ideally) a shared resource among a great many projects and is thoroughly isolated from the application code of those projects via the firewall of the UIDL parser.

Regrettably, the field of user interface design is no different from the rest of the computing industry with respect to the problems of technology time lag. Hardware advances have been occurring at a dramatic rate and with them, the potential of individual machines to devote greater resources to user interface issues has sparked great interest and invention with respect to the styles and paradigms for man-machine interaction. As ideas develop into prototypes and prototypes into practice, tools to streamline the development process often lag well behind the
state of the art. The practical upshot of this all too familiar situation is that we are now placed in an environment where we have excellent, efficient and robust tools for creating what, in the user's eyes, is seen as an archaic user interface; we have maturing tools which do an adequate but less than ideal job of creating what a user would consider a 'standard' graphical user interface; and, we barely have any high level tools that can begin to address the problems of creating state-of-the-art user interaction environments.

Unlike many previous increments in the evolution of user interaction styles, the transition from a WIMP to a non-WIMP interaction paradigm offers little opportunity to capitalize on existing UIMS tools and technology. Traditional user interfaces have been built around five basic assumptions regarding their structure and operating environment, non-WIMP interfaces invalidate these assumptions and with them the tools that supported them. A table of the contrasts between the two interaction paradigms is shown in Figure 1-1.

<table>
<thead>
<tr>
<th>Traditional User Interfaces</th>
<th>Non-WIMP Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-threaded I/O stream</td>
<td>Parallel, asynchronous, interrelated dialogs</td>
</tr>
<tr>
<td>Discrete input tokens</td>
<td>Continuous and discrete inputs and responses</td>
</tr>
<tr>
<td>Precise tokens</td>
<td>Probabilistic input, possibly difficult to tokenize</td>
</tr>
<tr>
<td>Sequence of events, rather than timing of events, is considered</td>
<td>Real-time operating requirements, absolute deadline based computations</td>
</tr>
<tr>
<td>Explicit user commands</td>
<td>Passive monitoring of user</td>
</tr>
</tbody>
</table>

**Figure 1-1:** The contrasts in the underlying assumption between WIMP and non-WIMP interfaces.
Exploitation of the basic assumptions for WIMP interactions has allowed traditional UIMS designers to capitalize on existing compiler technology, particularly in the areas of parsing UIDLs and managing the dialog stream between the user and the application [12]. While the specific details of lexical analysis may vary greatly with the overall interface design, once the input stream has been tokenized, the syntax and semantic meaning of the WIMP style dialog can be readily described using well established compiler technologies such as Backus-Naur Form (BNF) or Augmented Transition Network (ATN) syntax specifications [9]. By violating the basic assumptions of traditional interfaces, the non-WIMP paradigm has not only limited the scope of existing tools available to support it but also brought into question whether application of compiler-based technologies themselves are an appropriate approach to the problem of creating new tools.

THE SPECIFICATION PROBLEM

The first objective which must be addressed in the creation of a UIMS for non-WIMP style interfaces is the creation of a specification language which is well suited to the task describing the human computer interaction dialog itself. The term 'well suited', while subjective in nature, is used to differentiate approaches which are mathematically possible but impractical from those optimized for the task at hand. Thus, while hard coding an entire virtual reality front-end from scratch in C or assembly language does constitute a solution to the interface specification problem (and this solution is all too prevalent in the industry today) it is far from an ideal abstraction for the interface designer.
Before a high level abstraction can be formulated, the basic structure of non-WIMP interactions must first be identified. Clearly, discrete events continue to be an issue, just as they are under WIMP style interaction. This paradigm is broken, however, with the introduction of continuous, parallel relationships between elements of non-WIMP, virtual environments. These relationships constrain both the behavior and presentation of interaction objects within the environment and may also result in cascade effects resulting from actions which are initiated by the user but are not continuously under the user's direct control.

For example, consider a virtual room where gravity, elasticity and momentum govern the passive behavior of a virtual ball. A user entering the room may grasp the ball and throw it against a wall. The ball will bounce in accordance with the environment's approximation of physical laws and the user may elect either to watch the ball ricochet about the room or to attempt to catch it. While this is, from a virtual reality perspective, a trivial example, it illustrates the different classes of interactions encountered when creating a direct manipulation environment. The force of gravity and laws of elasticity and momentum that govern the trajectory and bounce of the ball are permanent constraints on the passive elements in the room. The user's grasp operation is a discrete event which results in the creation of a continuous but temporary relationship between the position and motion vector of the ball and the position and motion vector of the user's hand. The release of the ball is likewise a discrete event, this time signalling the end of the relationship between the hand and the ball and locking in the initial parameters of the ball's trajectory. The existence and concurrent enforcement of the generic constraints allows the ball to continue to fly (and potentially interact with other elements of the environment) without any further action on the user's part.
In this context, the objective of developing a specification language for non-WIMP interactions becomes the task of devising a system which is well suited to describing large systems of parallel, continuous relationships, some of which are permanent constraints while others are activated and deactivated in response to discrete events within the environment itself.

In addition to the problem of devising a specification language to accommodate the basic nature of the non-WIMP interaction paradigm, software engineering concerns must also be addressed. User interface implementations are notorious for requiring revisions, especially in the early stages of a product's life cycle before its user community has stabilized. The ability to maintain and evolve the user interface at a high level of abstraction can become a crucial aspect of the product's overall viability.

Software engineering concerns consist of a great many factors, four of which stand out with significant importance with respect to the problem of creating a specification system for non-WIMP style interfaces. These four are detailed below [92] [93]:

- **Scaleability**: Any solution to the specification problem must be able to address the broad spectrum of tasks to which it will be applied. The difference between an academic solution and a practical one often lies in its ability to scale up from a trivial proof of concept case to a full blown implementation without becoming overly cumbersome or unmanageable.
• Extensibility: Physical input and output devices continue to evolve as do the interaction paradigms that revolve around them. Any solution to the specification problem that makes assumptions about number or scope of I/O channels and their uses, runs the risk of becoming outdated with the invention of new equipment. A more viable solution should focus on providing a consistent framework upon which reusable interface modules to both current and future devices can be built.

• Modularity: As user interfaces become larger and more complex there is a need to be able to divide the interface specification into manageable parts which can be maintained with a high level of isolation with respect to the remainder of the system. This is particularly the case in areas such as immersive virtual reality where the volume and complexity of the code devoted to the user interface may dwarf that of the actual underlying application engine.

• Reusability: The ability to reuse stable, mature code units can result in dramatic savings with respect to development time and effort. Traditional UIMS packages usually address this issue by providing libraries of commonly used interaction objects, or toolkits, which can be used by the user interface designer in a black box fashion. This fixed widget set approach to the reusability problem may not be appropriate to the non-WIMP situation, however, in that current research and practice has yet to determine what the set of generic fundamental interaction objects consists of for the non-WIMP paradigm.
THE SHADOW SYSTEM

The remainder of this work details the design, implementation and application of a user interface management system and its associated user interface description language. This UIMS, the SHADOW System, was developed specifically to address the many unique issues which arise when trying to work within the domain of non-WIMP interaction styles with particular attention paid to the demands of direct manipulation virtual environments. Unlike many previous efforts in this area [1][49][52][61][62][70][104], the SHADOW System was designed to be able to support full scale development efforts of meaningful, maintainable, and complete user interfaces rather than simple proof of concept exercises and, as such, considerable effort has been expended to explore and integrate many principles from accepted software engineering practice.

While the primary thrust of this work was to create a system suitable for the specification, generation and execution of non-WIMP style user interfaces, care has been taken to provide a certain level of applicability to the needs and issues of traditional WIMP style interfaces as well. Additionally, the SHADOW paradigm was built around the idea of extensibility and makes a deliberate effort to avoid architectural restrictions on the numbers and types of input and output channels available to the user interface designer and strive to avoid assumptions regarding the classes of interaction problems to which it may be applied.
SECTION II: RELATED WORK

INTERFACE DESIGN CONCEPTS

In 1974, J. D. Foley and V. L. Wallace proposed that the design of a user interface could be partitioned into a hierarchy of specification details [32]. The proposed scheme consisted of four design levels: Conceptual design, wherein the basic entities of the application and the attributes and interrelationships of those entities are defined; Semantic design, wherein the functional requirements of the application are addressed by the creation of a command set which governs the processing of conceptual entities; Syntactic design, wherein the protocol for accessing the command set and presenting status information is defined; and, Lexical design, wherein logical tokens are bound to specific, low level hardware-based interactions.

Interaction styles have evolved considerably in the past twenty five years [125] but the four partition task model continues to serve as a valid and valuable design model. At its heart, the proposed scheme recognizes the fundamental difference between designing what an application does (conceptual and semantic information) versus how a user accesses those facilities (syntactic and lexical details). By partitioning these design tasks, the Foley and Wallace scheme actively encourages the design team to separate the core application processing tasks from the more volatile interface elements and lays the foundation for user interface design as a separable and independent (but interrelated) development effort.
This separability gave rise to the concept of dialog independence [45]. H. R. Hartson and D. Hix propagated this concept as a direct analogy to accepted data independence techniques and sought to encourage the formal decoupling of interface related elements from core application algorithms. This formal decoupling was found to provide several advantages over the monolithic application designs which had been commonly employed in the past. From a design and development standpoint, a formal separation allowed the interface design team the freedom to use specialized tools and development techniques which simplify interface building tasks but may not be suitable for the development of an entire application. Additionally, decoupling an application’s front-end from its underlying processing engine enables the core application to support multiple user interfaces (or evolvable interfaces) without incurring costly redesigns of the base operations.

At the time of publication, as well as in many application domains currently in use, the concept and applicability of dialog independence was clearly defined. In a spreadsheet, for example, the core application is responsible for maintaining a database of numbers and formulae and for applying constraints as needed. The interface module is concerned with data entry, edits, and the mechanics of displaying results. The two parts communicate with one another via a simple procedural protocol. In the recent past, however, the emergence of direct manipulation and virtual reality interaction styles has blurred the interface versus application distinction. The question of where to draw the line between the concept of a ball which can be tossed in a model of the world and a virtual ball which is tossed by the user inside a graphic rendering of that model has yet to be resolved.
ARCHITECTURAL CONSIDERATIONS

In order to support accepted software engineering practice and the separation of programming tasks, several architectural models have been developed. J. D. Foley, A. van Dam, et al. describe a layered design approach [31] which strives to support portability and maintainability by introducing a system of software strata that buffers the application from platform-specific hardware details while adding increasingly sophisticated layers of abstractions. This design resembles established architectures used in computer networking and communication, and is illustrated in Figure 2-1.

![Diagram](image)

**Figure 2-1:** Implementation strata as proposed by Foley and van Dam.

Under this scheme, the application retains the ability to interact with any layer when needed, but is strongly encouraged to employ the services of the highest layer which is capable of completing the task at hand. Thus, most interactions should be handled by the UIMS via abstract events, callbacks, etc. Where functionality is required but not available within the
confines of the management system, the application may resort to accessing toolkit, graphics
system or operating system services using increasing complex and low-level service call
libraries. Ultimately, if no abstraction has been provided for a given task at any software level,
the application may directly access the physical capabilities of the platform directly and takes on
the responsibility (and complexity) of hardware management and bit-by-bit data manipulation.
Ideally, the practice of coding directly to the hardware layer should be avoided and, when
unavoidable, highly isolated from the balance of the application.

Like the four-level design proposal, this layered architecture deals in generic abstractions and is
more or less independent of the interaction style being implemented. It does not propose any
particular specification language or programming paradigm but its design is strongly influenced
by traditional, third generation language implementation practice.

In 1985, a collection of user interface system experts at a conference in Seeheim, Germany
developed and proposed the Seeheim Model [38] as a potential application architecture. This
system is strictly built around the concepts of dialog independence and four-level design and
strives to codify the modular separation of the underlying application and its interface. Under
this model, the front-end of an application is divided into three components: Presentation,
which deals with lexical issues of input and output; Dialog Control, which addresses issues of
syntax; and the Application Interface, which addresses issues of data sharing with the
underlying core functionality. The elements of this model are strictly layered such that the user
may only interact with the presentation layer. Presentation may interact with dialog control but
has no access to API functionality. The dialog control unit may call upon the services of the
presentation layer or the API but has no direct channel to either the user or the core application,
etc.
This strict separation has both good and bad qualities, depending on the nature of the work being addressed. The concept of token passing between distinct layers allows high levels of isolation and specialization within those layers. This can be quite beneficial in the case of WIMP style interfaces where presentation elements such as scrollbars and buttons are commonly used and reused across a variety of applications. The weakness of the system surfaces when faced with the task of dealing with highly interactive systems where the strict layering may result in redundant functionality and excessive overhead, and in certain forms of non-WIMP applications, where the lexical signals may be probabilistic and difficult to tokenize without contextual information not normally available at the presentation layer.

In contrast with the stratified Seeheim model, G. E. Krassner and S. T. Pope proposed the Model View Controller Architecture [85]. Under this system, the application is again divided into three base components: the Model, which addresses conceptual entities and underlying functionality; the View manager, which manages all of the heuristics for output presentation of the model; and, the Controller, which governs and constrains input operations.

While this design reduces the processing overhead and redundancy of the Seeheim model, it also eliminates any codified enforcement of the distinction between lexical and syntactic design issues and differs from the widely accepted four-level design paradigm opting, instead, for a simpler Input-Application-Output model. This architecture is not widely used in UIMS design but has been applied successfully for creating multi-agent models in Smalltalk.
SPECIFICATION LANGUAGES

With the rise of user interfaces as conceptually separable entities, the task of specifying and implementing those interfaces became a distinct problem in its own right and a variety of solutions has been proposed. These solutions draw from the knowledge and experience base accrued over the years in other domains within computer science, and each has been found to have particular areas of merit, as well as limitations.

The most popular and commercially successful approach to this problem is currently the use of toolkits. These solutions provide an interface development library of interaction objects (or widgets) which the application developer may access via a traditional third generation language. Typically these systems are event driven and use some form of callback mechanism or shared memory to exchange data between the underlying application and the front-end. Examples of such systems may be found in the Apple Macintosh Quickdraw library, Motif [141], and Interviews [90].

The toolkit approach allows programmers to capitalize on a large body of mature, stable, and robust interaction objects for performing commonly needed lexical operations without having to make radical departures from traditional design and coding paradigms. In addition, reliance on such toolkits helps to enforce a common look and feel across multiple applications. Unfortunately, this reliance also represents a limitation and a liability. If an interaction object is needed which is not supported by a given toolset, the designers must either work around the desired functionality or go to the often painstaking procedure of extending the widget set (if such is possible at all). Additionally, the burden of enforcing dialog independence is left to the
software engineers and the coding standards of the development organization. The potential for freely intermixing interface related code with core application algorithms is high, and has been shown to be detrimental to the long term maintenance of the finished application [92].

Formal grammars have also been used as the basis for specifying user interactions. The SYNGRAPH system [109], proposed by D. R. Olsen and E. P. Dempsey, was built around the use of a BNF grammar which had been extended to provide production rules for both accepting inputs as well as generating outputs. This is useful from the standpoint of codifying the nature of individual dialogs and has also proved useful in predicting user performance based on explicit tracing of lexical interactions [117]. Unfortunately, scaling a BNF specification has the potential to become quite cumbersome and difficult to trace. Moreover, as interfaces move further and further away from the rigidly defined realm of command line and event driven interfaces into the multi-threaded, probabilistic worlds of non-WIMP interfaces, the applicability of BNF structured grammars becomes questionable.

Automata theory has likewise served as a basis for specification languages in the past. In 1968, W. M. Newman's Reaction Handler [104] (considered to be the first example of a UIMS) added flow chart elements to simple state diagrams to generate user interface code capable of orchestrating simple textual dialogs as well as servicing light pen based selection interactions. In 1985, R. J. K. Jacob proposed a UIMS based on Moore Diagrams [70] which offered many of the same benefits as the BNF based approaches explored earlier but offered more traceability due to the visual nature of the language. Unfortunately, these systems also suffered from problems of scale and maintainability with little or no support for modular design
concepts. Likewise, the underlying token-based, event driven paradigm embraced by these
designs, may not be sufficient or optimal to satisfy the needs of non-WIMP interface
designers.

This is not to say that event driven approaches are universally inappropriate. Quite the
contrary, within the domain of traditional WIMP style graphical user interfaces, event driven
paradigms have been shown to be a robust and useful scheme of managing multiple, non-
sequential, discrete dialogs in an orderly and maintainable fashion. Several user interface
management systems, including ALGAE [27], Sassafras [50], and the University of Alberta
UIMS [39], have been constructed to exploit the basic paradigm of a centralized event loop and
a pool of event handlers that are evaluated in response to a particular event or sequence of
events. The ability to respond to discrete actions continues to be an issue within the realm of
non-WIMP systems and as such, the contributions of these proposals should not be
overlooked.

In contrast with the discrete event driven school of thought, constraint systems have also been
used in the domain of interface specification and development. Due to the time and complexity
questions of satisfying constraint networks, most applications to date have focused on issues
of simple screen layout or input data verification, however, entire user interface management
systems based on constraints have been built and demonstrated. Of particular note is the
RENDEZVOUS [52] system developed by R. D. Hill. This system proposed an Abstraction-
Link-View architecture which is similar to the Model-View-Controller scheme used by
Smalltalk but used a constraint programming approach to I/O handling rather than the pure
object oriented one advocated by Smalltalk. The use of constraints offers an ideal way of
modeling continuous relationships between entities, especially within a virtual world where the 
interface is tasked with the job of simulating real world phenomena. The problem, in this 
respect, becomes one of scale, performance and management to ensure that complex behaviors 
may be both expressed and applied in a way that is both reasonable for the programmer and 
fast enough for the user.

Surprisingly, most specification languages to date have elected to exploit only one particular 
class of solution. The one notable exception to this is the PMIW system [63] proposed by R. 
J. K. Jacob. This system suggested the use of a hybrid solution drawing on the fortes of both 
constraint-based and automata-based proposals to create a graphical user interface description 
language capable of addressing some of the more unique needs of non-WIMP interface 
designers. While more of a proof-of-concept proposal than a robust language, the PMIW 
system demonstrated that the concept of a hybrid solution was viable. Unfortunately, PMIW 
itself suffered from many of the defects of its predecessors such as an inability to scale, a 
closed architecture with respect to extensibility, little support for software engineering 
practices, and poor support for modular programming. Nonetheless, many of the specification 
concepts embodied by this system served as the founding principles upon which the SHADOW 
System was based, and as such PMIW represents the SHADOW System's most direct 
ancestor.
SECTION III: PROBLEM REFINEMENT

OVERVIEW

As discussed in Section I, the creation of a specification language suitable for designing and implementing non-WIMP style user interfaces gives rise to a number of issues and concerns not normally encountered in the course of developing traditional user interfaces. This section will isolate, identify and refine these issues so that they may be explicitly addressed within the design of the SHADOW System and so that the effectiveness of the SHADOW System to handle such concerns may be measured.

CONCEPTUAL CONTINUITY ISSUES

Within a uniprocessing digital computer, all CPU activity is, by definition, sequential and discrete. This hardware-based behavior, however, should not be allowed to dictate the conceptual level design of a given user interface. At the level of a conceptual specification, any given design should be virtually hardware independent, deferring the task of platform specific implementation to the UIMS. The emphasis of a UIDL which supports non-WIMP interaction should therefore be placed on ensuring that the language used captures and reflects the designer’s cognitive model of the interface’s intended behavior rather than the specific details of a particular implementation of the model.
Conceptually, processes in the real world may be seen as collections of both discrete events and continuous relationships. For example, in a game of billiards, the cue strikes a ball (discrete event) causing it to roll along a particular trajectory (continuous behavior) until it bounces off a rail or strikes another ball (more discrete events). In designing a computer simulation of this same situation, it is beneficial to be able to maintain this level of abstraction to increase the likelihood that the resulting implementation will reflect both the semantic intent of the solution as well as the syntactic details a specific solution.

In light of this, a non-WIMP UIDL should support specification mechanisms that allow interaction object behavior and manipulation to be described in both continuous and discrete terms. It should provide sufficient flexibility so as to allow the designer to draw from both domains concurrently and create hybrid specifications which closely align to his or her conceptual model of the underlying behavior being described. The task of translating these models into executable code should be the responsibility of a platform specific user interface management system, further detaching the user interface designer from the actual implementation details.

Similarly, many processes in the real world conduct themselves in parallel with one another. Some such processes interrelate, possibly precipitating new processes, while others proceed essentially independently of one another.

When attempting to describe behavior which is innately parallel and independent of other processes within a system, the designer should be both allowed and encouraged to isolate
process specifications with the understanding that behavior they represent will eventually be demonstrated concurrently. For interrelated processes, the nature of the relationship should be clear while details of synchronization and task rendezvous should be passed on to the level of implementation. The issue of concurrent execution on within uniproprocessing environment should be the responsibility of the UIIMS and the operating system and entirely detached from the user interface design process.

REAL TIME BEHAVIOR ISSUES

In many non-WIMP interaction styles, execution performance and timing may play pivotal roles in the overall success of a particular user interface implementation. In the case of immersive virtual reality, delays in the generation of visual stimulation may result in nausea for the unsuspecting user [24]. On many graphics platforms the frame refresh rate constitutes a fixed, periodic deadline and consistently meeting this deadline becomes the driving concern of the interaction environment. This is a philosophical reversal of the tenets of Computer Generated Imagery (CGI) used by the entertainment industry for animations and a departure from the typical approach taken by current software designers and algorithms in general.

In CGI animations, the objective is to produce the best image at all times regardless of rendering time involved. Each finished frame is captured and sequenced for playback to the viewer once the entire animation frame set is complete, allowing the final viewer to become oblivious to the hours expended to render a static sequence in exacting detail. In a related vein, current office automation and desktop game software designers have the luxury of a user
community trained to work in a half duplex environment with computers that never seem to be fast enough. When a dynamic output must be generated in real time, the designer will typically either generate it as fast as possible, knowing that if it takes an excessive amount of time the user will wait (often contemplating the purchase of additional memory or a faster motherboard), or give the user some form of warning marker (such as changing the cursor to a wristwatch or hour glass) and explicitly ask them to wait until the operation is complete.

Unlike CGI animation and current desktop software design, where the primary objective is to produce complete and correct output with little regard for time constraints, non-WIMP systems such as immersive virtual reality are often faced with the challenge of producing the best approximation of the desired output within the time available. Any specification scheme intended for use under such circumstances should be cognizant of these trade-offs and should provide an infrastructure through which the interface designer may specify both the ideal behaviour of the system as well as contingency plans and guidelines for decimation should a given hardware platform prove incapable of maintaining an acceptable throughput rate.

Within such a paradigm, the designer should once again be removed from the implementation details for a given platform and afforded a level of abstraction wherein the designer's responsibility is to provide the UIMS with an assortment of alternative algorithms of varying time complexity along with hints and preferences as to the optimal situations under which each alternative should be explored. For example, the designer of a virtual world might include a solid object of a brick building within the world. If time is available, the object is rendered with ray tracing to introduce correct contour shadowing etc. If time is tight, the building may be reduced to a texture mapped shape. If time is very tight the object might further be reduced
to a red box to serve as a visual place holder in the hopes that subsequent frames may be able to reintroduce a previously decimated level of detail. Similarly, the underlying processing algorithms of a system may be subject to decimation. A number crunching module seeking to integrate a given relationship might apply Simpson’s Rule to the equation if time is available, or use the computationally less demanding Trapezoidal Rule if time is running low. The frame by frame decision as to which algorithms to apply and the ability of the overall system to meet its output deadlines should be the responsibility of the UIMS itself.

The issues of real time management are further complicated by the introduction of full duplex I/O commonly encountered in many forms of non-WIMP user interfaces. Unlike traditional WIMP style graphical user interfaces (and most of the interaction styles which predated them), a lack of user input to a non-WIMP application does not necessarily translate into idle time. Traditional user interfaces have tended to rely on half duplex communication and simple stimulus-response programming. More often than not, 'idle time' is spent waiting for additional discrete input such as a mouse click or keyboard interrupt rather than performing useful processing (a notable exception to this observation can be found in many video games where continuous streams of output are generated with or without user input; but then, many video game interfaces can be classified as non-WIMP interactions).

In contrast, many non-WIMP systems do not have the luxury of simply polling for a particular discrete event and updating their output accordingly and, instead, must actively monitor continuous streams of data trying to identify probabilistic tokens of context-sensitive, semantic value. For example, non-command interfaces will often monitor the user using a variety of sensors such as cameras, radio-magnetic trackers and infrared detectors. Depending on the
resolution of the sensors, the user has the potential to be seen by the system as being continuously in motion yet few, if any, of the gestures and motions detected within a given period of time have semantic value to the system. While the process of monitoring the user and identifying tokens can (and in most cases should) be thought of as a separate process executing in parallel with the balance of the system, it is not an independent process in that it must be tightly synchronized with the processing of the core application threads, especially if elements within the core application exhibit autonomous or continuous behavior in the absence of user intervention.

A simple example which illustrates this point is the act of catching a ball within a virtual world. A virtual ball within the environment will continue to follow its trajectory whether the user is actively trying to catch it or passively watching it fall. The movement to catch the ball requires both the position of the hand in the path of the ball as well as recognition of a grab gesture at a time when the ball is within the forward reach of the fingers. If the synchronization between the positioning of the ball and the tracking of the hand gestures is in error, that ball may bounce off or pass through the avatar of the user’s hand even if the user’s eye-hand coordination was on target for catching the ball based on the data provided.

To address these complexities, a UIMS intended for use within this class of interface should strive to provide both a specification language that will allow isolation of user monitoring and data model updating activities with respect to cognitive scope while simultaneously providing a run time engine which is sensitive to the temporal correlation between these tasks. Ideally, the system should allow the user interface designer to use temporal abstractions to specify time management and synchronization guideline between otherwise isolated threads and leave the low level details of cycle utilization to the run time engine.
SCALEABILITY ISSUES

In sharp contrast to the early days of computer science, modern user interfaces have become large, complex subsystems often requiring more hardware resources than the underlying applications themselves. This is especially true in the areas of non-WIMP style interactions and all forms of graphical user interfaces. To be successful under such circumstances, a user interface management system should be designed to anticipate growth in interface complexity and application of its associated user interface description language to large scale developments. In addition to simple performance concerns, this requirement suggests that the design of the UIMS should be cognizant of formal software development processes and provide mechanisms to support such techniques.

The first hurdle to overcome in the creation of any large scale application is usually not a question of the machine's ability to execute a broad range of complex tasks but rather question of the human's ability to specify exactly what those tasks should be and how they should interrelate. In the context of system design, the cognitive load of a problem may be thought of as its breadth of context times its level of detail and represents the limits of the human mind to work effectively with a variety of diverse facts, constraints and objectives while maintaining a sense of perspective. A user interface management system intended for use in a large scale environment should be sensitive to cognitive load issues and provide facilities for dynamically scoping the breath and depth of complexity presented to or required from the designer at any given point in the design process. This allows the designer to selectively decompose complex areas of the system into simpler elements while maintaining a sense of how all elements
contribute to the overall application. Additionally, support for cognitive load management lends itself readily to modular design principles and in turn facilitates efforts at code reuse (discussed later in this section).

As with any software development (user interface related or otherwise), as systems grow in size they run the risk of becoming unmaintainable. While there are a great many contributing causes to this phenomenon, one of the simplest is a lack of traceability; the control and data flow threads through the system cease to be obvious and more time and effort is spent identifying where modifications should be made rather than actually enhancing the code. A user interface management system should acknowledge this risk and strive to mitigate it. Possible approaches to the traceability problem might include embedding reverse and re-engineering tools within the UIMS and/or using a description language which is itself inherently graphical, making data flow, control flow, and hierarchical dependency visualizations readily available to the designer and maintainer.

In interfaces which are event driven by nature, as the breadth of a user interface increases, so does the number of token event producers and consumers. This raises two additional issues which need to be addressed. First, the output of event producers must be propagated to the event consumers. Second, a consumer who has no vested interest in a particular event should not be alerted to the generation of such an event. This is to say that for any given class of event being generated, the event should be propagated to exactly those consumers who have an active need for the given information. For example, a widget designed to operate in response to mouse clicks which has no provisions for responding to keystrokes should never be alerted to the presence of keyboard input.
Additionally, a given class of input events has the potential to be subdivided by locality, especially in the case of large, centralized multi-user applications. Under such a scheme, each user might possess an identical set of input devices and is shown an identical user interface. When input tokens are generated, the application needs a mechanism to associate each event with the user who precipitated it and to propagate the event exclusively to the user interface elements available to that user.

To address these concerns, a user interface description language designed for large scale applications should provide mechanisms to allow the designer to discriminate between event based both on the class on event as well as the event's point of origin. Additionally, the UIDL should allow the designer to provide hints and directives to control the generation and propagation of events throughout the system without becoming bogged down in the low lying complexities of event management.

**EXTENSIBILITY ISSUES**

Compared to other interaction styles, the domain of non-WIMP user interfaces is still in its infancy and is continuing to grow and evolve in unpredictable directions. Any UIDL targeted at specifying interfaces within this domain must be flexible and extensible enough to adapt to these new twists of evolution or risk being outgrown by the very interaction style that motivated its own existence.
Three areas within the non-WIMP interaction style domain which have demonstrated the most serious volatility to change are the handling of input devices, the management of new output channels and the introduction of new interaction objects (often termed widgets). Even traditional user interfaces which have the luxury of (relative) standardization with respect to the handling of keyboards, mice and raster graphics displays are often ill-suited to deal with some of these situations. For example, a digitizer tablet and stylus is a form of (X,Y) pointer input device that may be used in place of a mouse in a traditional WIMP interface; however, anyone who has ever tried to sign their name using a mouse can verify that a digitizer tablet is a very different class of input device with capabilities that most current software does not exploit.

Alternative input devices present a myriad of issues which the user interface designer must recognize and strive to address. Such issues include defining the taxonomy of capabilities represented by these devices, acknowledging the individual fortes of each device for performing particular classes of actions and managing the unique protocols needed to communicate with them. Where possible, user interface management systems should be sensitive to these issues and provide mechanisms for dealing with the introduction of new devices in a manner consistent with the overall structure of the UIDL. Historically, many systems have ignored the unique capabilities of particular devices and have taken the much simpler approach of defining a base set of generic devices which appeal to the lowest common denominator. For example, many user interface management systems assume a pointing device to be a pollable (X,Y) input channel with multiple buttons which are either pressed or not. This solution provides a strong level of isolation between the application designer and the input devices actually being used by the end user. For traditional WIMP interfaces, this separation is usually desirable in that mice and track balls are logically equivalent with respect
to their ability to select menu options and click on screen buttons. Applications which need functions beyond those provided by the generic UIMS pointer and keyboard device are comparatively few in number and are often left to fend for themselves with respect to accessing the specific features of the devices in question. Within the domain of non-WIMP user interfaces, however, alternate (and sometimes custom) input devices are common. For a UIMS to effectively support such interfaces it must provide a mechanism by which such devices may be defined and integrated into the UIDL smoothly, consistently and in a reusable fashion.

Similarly, many non-WIMP (and even state of the art WIMP) interfaces demonstrate a need or desire to explore output channels beyond traditional bit-mapped video and audio tone generator. Unlike these common venues, whose features are usually readily accessible via well defined programming libraries and APIs, many output devices such as tactile feedback systems and speech synthesizers often do not have the benefit of robust, high level support libraries, have interfaces which are constantly in a state of flux due to revision and evolution of the underlying hardware, and/or are accessed via custom or proprietary logical protocols which vary widely from one vendor to the next despite similarities in the physical functionalities in the devices. A user interface management system for non-WIMP systems should be cognizant of these issues and provide facilities both for integrating alternate output devices into the logical scope of the UIDL while at the same time buffering the interface specification from the volatility of the low level interfaces to the physical devices. As with the case of input devices, the syntactic handling of new output channels should be conceptually consistent with more traditional ones within the overall specification language.
The existence of new input and output devices gives rise to the definition of novel interaction objects. Traditional WIMP toolkits usually consist of a base set of well defined objects (or widgets) which serve as the basic building blocks for all interfaces developed under the given system. As in the case with input and output devices, the use of a pre-cast widget set is both a benefit and a limitation for both the given toolkit and any UIMS or UIDL based upon said toolkit. Applications which require interactions lying entirely within the scope of the widget set may be developed quickly, easily and consistently while those few applications which need additional mechanisms outside the base set anticipated by the toolkit designers are forced either to break with the bulk of the toolkit's development paradigm, or to extend the widget set to include the desired interaction object. Many existing toolkits provide mechanisms (some more obscure than others) to pursue the latter option, unfortunately, many user interface management systems based on such toolkits do not support automation of this activity, further convoluting the UIMS-based interface development process.

The novelty and varied nature of non-WIMP style interfaces has, to date, precluded the definition of a flexible and complete widget set across all domains. This is particularly true in the area of virtual reality where the environment itself is an interaction object after a fashion and traditional, accepted interface elements such as pull down menus and text fields appear conceptually out of place, artifacts of an abandoned personal computer desktop. In light of this, it seems unlikely that a user interface description language suitable for non-WIMP style interfaces can be developed within the confines of a predetermined set of interaction objects and that, instead, the emphasis should be placed on ensuring that both the UIDL and its associated UIMS provide facilities for the definition, implementation and reuse of highly customized interaction objects allowing the application scope of the UIMS to be readily extended without compromising backwards compatibility.
MODULARITY ISSUES

The need for scalability and extensibility suggest that any specification language suitable for application within non-WIMP domains will face many of the same issues as any high level programming language. Foremost of these issues is the need to provide facilities which support functional decomposition of user interface elements, decoupling of abstract processing threads and the application of modular design principles. Language features which support these goals include provisions for data hiding, encapsulation, data abstraction and, in the case of a user interface context, event abstraction.

When attempting to manage the cognitive load of a design problem or implementation, the common approach is to reduce the level of detail displayed with the current scope in order to increase the breadth of view and vise versa. This approach relies on the realization that not all data elements and routines within a particular functional area are created equal with the context of the greater application. Some items are used exclusively for internal (local) operations while others form the interfaces by which the functional areas interact with the balance of the system. When the need arises to broaden the context in which a functional area is viewed, the internal details may be hidden from the designer/programmer to free up space for new information to be entered into scope.

The conceptual masking of low level details is most effective when it is paralleled by a lexical enforcement mechanism within the specification language itself [35]. This is to say that once the scope of data items has been determined from a conceptual standpoint the language should
provide facilities to ensure that logical access to these items has been restricted accordingly, reducing the likelihood of accidental modification or ambiguous referencing. Under such scope management, data and implementation details hidden from higher perspectives are selected not simply to prevent cognitive overload, but because the information they represent has no lexical, syntactic, or semantic meaning at the higher level.

Similarly, government and commercial studies have suggested that when certain data elements and the operations on those elements are tightly coupled together from a conceptual standpoint, it is beneficial to formally couple their implementation so that the data and it direct manipulators may be treated as a single unit with respect to cognitive load management [8][92][93]. This encapsulation is one of the primary tenets of object oriented programming.

Historically, an object oriented approach to interface specifications has been quite successful, especially with respect to graphical user interfaces where widgets like push buttons and labels lend themselves readily to being represented by an object model [141]. While the situation with non-WIMP style interactions is far less defined with respect to what a widget is let alone how it should be modelled, experience with such systems as virtual world simulations suggests a model featuring data hiding and encapsulation is a reasonable approach.

In addition to bundling data manipulation methods with their target datum, it is common practice in many high level languages to tightly couple related data items together into composite records which may be manipulated as singular units. The ability to create and manipulate these data abstractions again helps to alleviate the problem of cognitive load and ensures that tightly related data atoms will remain mutually synchronized within the record. In
a domain as diverse and volatile as non-WIMP user interface development, the ability to define, manage and interface with abstract data types within the user interface description language may be crucial to the success of language itself.

Extending the concept of data abstraction, any UIDL intended for use in the non-WIMP domain will most likely need to support some form of event abstraction as well. Many input and output devices currently in use and in development may be seen (in whole or in part) as event driven, usually at a very primitive, atomic level. Forcing a user interface designer or programmer to explicitly address these low level, detailed events at all tiers of the overall interface abstraction represents a lack of parallelism with the philosophy of cognitive load management and hierarchical modular design. In light of this, it is reasonable to introduce the concept of event abstraction, where primitive, generic event tokens may be grouped, correlated and/or sequenced by a succession of handlers to produce higher level, semantically meaningful events optimized for the processing at hand.

REUSABILITY ISSUES

Like many creative endeavors, early investigations in the field of non-WIMP style interactions have been focussed more around demonstrating a proof of concept than production viability. While this heritage is common to nearly all domains within computer science, the development process of non-WIMP systems has been slow to evolve away from it and many system continue to be built from scratch in the absence of support tools and standardized libraries\(^1\). In many cases this represents a wasteful duplication of effort and, where possible, should be avoided.

\(^1\)The one notable exception to this being in the area of 3-D rendering engines which have gained a higher level of refinement and application within the domains of virtual reality, CGI generation, and Doom-style video games.
Unfortunately, the issue of code reuse is not one that can be directly resolved merely with the inclusion or exclusion of a given set of language features within a UIDL. At best, a specification language can provide facilities which will allow code reuse efforts to be successful. It is unrealistic to expect an implementation or specification language to dictate the level of discipline, forethought and creativity of human designers and programmers. To a great extent, the success of code reuse efforts is a direct function of the people, processes and priorities of the development organization. Should such an organization desire to exploit the benefits of code reuse for the creation of standardized libraries, widget sets, etc., however, the UIDL in question should be accommodating to their efforts, allowing embedded documentation within specifications, modular isolation and rigid decoupling, and the introduction of customized lexical artifacts which permit the developers to define and impose rigid coding and naming conventions.

One of the key elements to reuse efforts is that code be well documented both externally and internally. While external documentation is beyond the scope of the specification language it is reasonable to require that the language provide some mechanism to allow the designer or programmer to annotate the software product internally. While this feature may seem trivial, it is often omitted from many commercial UIMS offerings for current WIMP systems under the (unsubstantiated) claim that the nature of the system is self-documenting and that human oriented comments would be redundant. Studies done for the United States Department of Defense have found that in many cases these claims are overstated if not false and that even in the well explored domain of WIMP interactions, embedded documentation linking syntactic expression with conceptual intent is beneficial. In the less mature domain of non-WIMP interactions it seems reasonable to assume that the value of such comments only increases.
While the government's investigations into GUI builders and its fifteen years of experience with the Ada programming language serve to undermine the myth of self-documenting code, this does not mean that coding standards and naming conventions are a waste of effort. Quite the contrary, they form a significant piece of the infrastructure needed to develop maintainable, reusable software elements. In this light, any UIMS which supports facilities for extension, scaling and abstraction should also provide either structured and enforced coding standards and naming conventions or sufficient flexibility with respect to the above to allow the development organization to impose its own. Given the level of diversity and relative lack of maturity with respect to the acceptance of standards within the non-WIMP domain, the former approach runs the risk of being too restrictive and limiting with respect to evolution while the latter leaves the system open to abuse by parties with no immediate interest in disciplined interface development.

The final element with respect to successful reuse efforts lies in the ability to isolate and decouple specific elements such that they become independent of their original contexts. This is in essence a logical extension of modular design. To support this goal, the UIDL should ensure that whenever coupling occurs between elements, the nature and extent of the coupling is both explicit and obvious. The language should be structured to minimize if not eliminate side effects while supporting a scheme for defining formal interfaces between subelements. On the other end, the language should allow previously developed elements to be introduced into ongoing efforts with minimal regard for the internal workings of the reused elements themselves.
INTELLECTUAL INVESTMENT ISSUES

For many of the reasons stated above, traditional tools and compiler based technologies are ill suited for addressing the task of creating non-WIMP user interfaces. Furthermore, this implies that any solution to the specification problem has the potential to obviate the knowledge and training user interface designers have already invested in the existing classes of tools and technologies. While it is unreasonable to expect significant innovation to occur without impacting the status quo, it is equally unreasonable to produce innovations which are so radically different from the norm that the intended user community cannot relate to them. In this light, an optimum solution to the specification problem should make every effort to exploit existing notations and conventions (where appropriate) to produce a schema which, while new, is not entirely foreign to its intended audience.

CLOSING COMMENTS

The following sections detail the origins, formal constructs and capabilities of the SHADOW System, a UIMS designed to address much of, if not all of, the design criteria discussed here. The SHADOW System itself consists of four major elements: SHADOW Talk, a graphical user interface description language presented in detail in Section IV; SHADOW Script, a textual UIDL equivalent for SHADOW Talk used internally by the UIMS and detailed in Appendix A; a compiler, which translates the SHADOW Script specifications into a machine executable form; and, a run time engine (described in Section V) which provides task and data management services to the generated user interfaces.
SECTION IV: THE SHADOW TALK MODEL

OVERVIEW

The task of creating a specification language to meet both the logistical constraints of non-WIMP user interfaces and practical software engineering concerns is a formidable one, especially when trying to balance the value of intellectual investments in less than optimal solutions. However, the difficulty of this undertaking can be reduced somewhat by capitalizing on the large body of related and semi-related work in the fields of automata theory, digital design, visualization, and declarative programming as well as general concepts borrowed from electrical engineering and computer science rather than focusing on the more limited scope of GUI designers and graphical toolkits.

Augmented transition networks have frequently served as the basis for describing user interfaces throughout the long evolution of interaction styles. One of the first such applications of networks was proposed by William Newman in 1968 [104]. The transition networks detailed in this work consisted of a hybrid between state diagrams and flow charts and, while originally targeted at interaction styles which might be considered primitive by current standards, many of the basic concepts embodied by the specification technique are directly applicable to any discrete event driven paradigm.

The use of transition networks for the specification of user interfaces has both desirable and undesirable consequences. State transition diagrams are a presentation media common to many
fields of study and as such may afford the beginning designer a level of familiarity with how to read, understand, and create them. Further, when used to describe grammars or dialogues, transition diagrams tend to be more readily traceable than textual descriptions (such as BNF) of the same information. The primary disadvantages of the graphical approach to the specification are the possibility of visualizations becoming over complex and cluttered as the information content grows beyond trivial cases and the fact that current computer technology is ill-suited to using the graphical specification directly and an intermediate (presumably machine generated) representation of the information must also be devised.

Precedent has also been created for addressing the non-event driven aspects of direct manipulation environments. The question of modeling continuous relationships between data elements, similar to the relationships exhibited by non-WIMP interfaces, has been addressed textually in declarative programming languages such as those used within spreadsheet applications to enforce one way constraints between the numerical values of cells. From the standpoint of a graphical specification, the field of electrical engineering has been using one way constraint networks for decades in the form of combinational circuit diagrams. While the data types available to the designer in a circuit diagram paradigm is extremely limited (on/off), the basic concept can easily be extended to more elaborate structures within a software forum.

Lastly, (and most significantly in the derivation of this work) Robert Jacob proposed a system for addressing the specification problem of non-WIMP interfaces in 1994. The PMIW specification scheme uses a coupling of visualizations to describe the behavior of a user interface. A state diagram is used to govern the handling of discrete events and a data flow graph is used to enforce continuous constraint relationships. The PMIW serves a strong foundation for the development of a specification language and user interface management
system for non-WIMP style interfaces but is not, by itself, a complete solution to the problem. The system was proposed as a concept and simple demonstration of technique and, as such, was lacking refinement, especially with respect to addressing software engineering issues such as modularity and scaleability.

The SHADOW System, is a direct descendant of the concepts and techniques embodied by the PMIW system proposed by Jacob [63]. Unlike PMIW, however, the SHADOW System has been structured to address many of the software engineering concerns discussed in Section III. The SHADOW UIDL is not, however, a direct superset of the one proposed for PMIW. Several changes were incorporated into the base design of the specification language to make it consistent and compatible with features needed to allow for the creation of full, robust user interface specifications.

Like its predecessor, the SHADOW System uses a visual specification language to describe the user interface. This visual language consists of an augmented transition network which addresses the handling of discrete input events, and a dynamic data flow graph to describe a network of one way constraints. Unlike PMIW, however, the SHADOW UIDL, SHADOW Talk, provides direct support for a modular design, hierarchical object containment, data and event scoping, and order of precedence within an inherently parallel and continuous evaluation environment.

Within the SHADOW Talk paradigm, a complete application is referred to as a system. A system, in turn, is comprised of three major components: one or more subsystems which describe the behavior of individual objects within the user interface environment; a (potentially
empty) generic constraint library which allows the user interface designer to specify rules and behaviors which transcend the individual objects' model of the world; and, a library of C++ code which constitutes the underlying, dialog independent application.

SUBSYSTEM STRUCTURE

The basic programmatic module of the SHADOW Talk language is called a subsystem. Conceptually, subsystems are designed to address the software engineer's needs for partitioning and encapsulating code into predominantly self contained, manageable blocks (the same role filled by object classes in C++ and packages in Ada). Internally, a subsystem is divided into three major sections: a public information interface, which describes the available channels the subsystem may use to interact with other subsystems; one or more directed data flow graphs which embody a set of continuous constraints used to transform the subsystem's input stream into its eventual output; and, an augmented transition network which dynamically restructures the topology of the data flow graph in response to discrete events. These divisions are illustrated in Figure 4-1. Each of the components is described in detail below.

![Diagram](image)

**Figure 4-1:** The three major components of a SHADOW subsystem specification
The SHADOW Talk language uses a containment paradigm for managing nested subsystems. Within this paradigm, subsystems within the same container system, or supersystem, may exchange continuous data and discrete event signals. In order to effect strong modularity and isolation between subsystems, each subsystem must define a public information interface which constitutes a firewall between the underlying operation of the subsystem and external signals and data. The interface specification itself consists of publicly accessible variables and an event mask which is used to restrict the breadth of event signals that the internal elements of the subsystem will be exposed to.

The public variables themselves are also partitioned into categories depending on their purpose and context within the subsystem. Variables used exclusively for providing values to the subsystem are called Stimuli and have read-only access within the subsystem. Similarly, strict output variables are termed Responses and have write-only access internally. The public variables in the third and final class are called Properties and are available for read-write access both internally and externally to the subsystem.

**CONSTRAINT MANAGEMENT**

The operation of computation within a subsystem is specified by its data flow graphs working in conjunction with its augmented transition network. The collection of flow graphs is referred to as the subsystem's plugboard. The plugboard is a graphical representation of a network of one way constraints that must be continuously satisfied to generate appropriate responses to stimuli in any given situation.
The continuous nature of the plugboard's constraint network by itself works well for enforcing permanent relationships between user interface elements (such as the correlation between the position of a steering wheel and the angle of a simulated car's front tires) but is insufficient to address the frequent need for temporary relationships (such as the direct connection between the position of a mouse pointer and the location of an object in the process of being dragged about the screen). To address these needs, each subsystem possesses an augmented transition network which defines how the topology of the data flow network should be modified in response to discrete events, such as a mouse button being depressed or a grasp gesture being realized.

Taken in tandem, the plugboard and its governing transition network provide a powerful user interface description paradigm with a strong visual orientation. This paradigm is well suited for both WIMP and non-WIMP style interface specifications. Each of these components is described in detail below.

The subsystem's data flow graphs, or plugboard, is a graphical representation of a constraint network. The term plugboard stems from the network's strong conceptual similarity to a digital logic prototyping board. This analogy is both deliberate and encouraged. Combinational circuits are, themselves, examples of constraint networks and existing research and user knowledge of their functionality and visualizations may be readily exploited to serve as the basis for the graphical programming environment of the SHADOW Talk plugboard component. As a result, finished plugboard designs frequently resemble wiring diagrams in both layout and style.
The plugboard itself is made up of links and variables. The links are the primary computation engines of the SHADOW System and are used to enforce constraints, to interface with external processes and to generate events. Variables are information conduits and short term repositories and are used to pass information from one link to the next within the plugboard, to interface with other subsystems, and/or to pass information to the related augmented transition network of the subsystem.

**LINK ELEMENTS**

A link is the fundamental processing mechanism available to the programmer within the SHADOW System and is somewhat analogous to the concept of a procedure in Ada or Pascal. Within the SHADOW System, the programmer specifies a procedural body for the link using either a subset of the C++ language which may include calls and class objects from external libraries, or a reference to an external subsystem which is to be encapsulated within the link's body, allowing subsystems of greater complexity to be built from simpler building blocks. The system is responsible for all call frame management and has the sole responsibility to deciding when, if at all, the link's body will actually be exercised.

In addition to providing a static description of the link's processing body, the programmer is also responsible for providing the UIMS with guidelines as to how the link should be treated with respect to its role in a dynamic constraint environment. These guidelines specify an operational context for the link which must be satisfied before the link may be activated. Such a context consists of: explicit flags under the programmer's control through the associated augmented transition network; polling priority requests which allow the programmer to indicate
the relative importance of one link's contributions to the constraint network over those of its peers; and, decimation hints which allow the programmer to specify how the system should treat the link with respect to limited processing time in the face of poor run time performance and fixed deadlines.

There are five categories of links available for specifying user interface functionality. First, links may be segregated by function. There are three divisions within this view. Input links are used to query device drivers and to read user action data such as mouse position or eye tracking information. In contrast, output links are used to activate and manage output channels such as displays, speakers, and tactile feedback devices. The SHADOW System performs all of its user interaction through the use of these input and output links classes. Furthermore, the number and style of I/O links is not restricted by the system, allowing new I/O links to be built at will and new forms of I/O devices to be integrated into the UIMS in a seamless and consistent manner. The third functional division for links is the data processing link or constraint object. This is where actual computation activities are specified and external processing may be initiated and is by far the most common class of link encountered within interface specifications.

In addition to segregating links by function, they may also be divided on the basis of their activation class. All links fall into one of two activation classes: dynamic links, which are conceptually continuous in nature and have the potential to be re-evaluated every time their output value is needed; and demand links, which are reserved for discrete processing and may only be evaluated as a result of a state transition within the subsystem's governing automata (similar to callback functions in traditional GUI builder systems).
The evaluation of dynamic links (the most common family of links in a non-WIMP style interface specification) is managed by the SHADOW UIMS and is primarily governed by a system of activation flags. Each link is associated with exactly one flag (identified by name). A given flag, however, may be associated with several links within the scope of the subsystem. A link becomes a candidate for evaluation (but is not immediately evaluated) when its activation flag is set to true. Whenever the output value(s) of the candidate links are required by the UIMS, the required links are evaluated using the most recent input data.

Demand links are more restrictive than dynamic links with respect to activation conditions. Normally, activation flags are set and cleared by state transitions within the subsystem's transition network. Once a dynamic link has been activated, it remains a candidate for evaluation until a state transition occurs which results in entering a state where the associated activation flag has been explicitly reset. In the interim, the active links may be evaluated and re-evaluated as many times as needed by the UIMS.

In contrast, a demand link will be evaluated at most once per activation. When a state transition occurs which explicitly sets the link's activation flag the demand link becomes a candidate for evaluation just as the dynamic link. Once evaluated, however, the demand link implicitly resets its own activation flag to prevent future re-evaluations until future state transitions explicitly re-activate it. This behavior is designed to service the needs for discrete responses within the the traditional WIMP style paradigm where a user initiated event results in the singular activation of a callback function to the underlying application. Additionally, demand links may be useful for

---

2A detailed discussion of link evaluation and the SHADOW UIMS run time behavior may be found in Section V: The SHADOW Run Time Engine
managing non-continuous output signals such as an audible tone or the playback of a pre-recorded message which, unlike a screen refresh, should not be repeated to the user every tenth of a second.

A complete matrix of the available link classes is presented in Figure 4-2. This table shows the purpose, classification and graphical representations of each of the five link families. Note that there is no specification for a demand input link. While, technically, such a class of links could be both created and evaluated, current investigations and applications of the SHADOW paradigm has yet to find a practical application for such an element.

<table>
<thead>
<tr>
<th>Activation Class</th>
<th>INPUT</th>
<th>OUTPUT</th>
<th>PROCESSING</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic</td>
<td>Activation Flag</td>
<td>Activation Flag</td>
<td>Activation Flag</td>
</tr>
<tr>
<td></td>
<td>Used to interface with</td>
<td>Used to manage continuous</td>
<td>Used to enforce continuous relationships</td>
</tr>
<tr>
<td></td>
<td>input device driver</td>
<td>output channels</td>
<td>within the user interface</td>
</tr>
<tr>
<td></td>
<td>processes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Demand</td>
<td>Activation Flag</td>
<td>Activation Flag</td>
<td>Activation Flag</td>
</tr>
<tr>
<td></td>
<td>Used for discrete</td>
<td></td>
<td>Used for discrete event callback processing</td>
</tr>
<tr>
<td></td>
<td>output signals such as</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>auditory error tones</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4-2:** The matrix of link classes and their graphical representations
Regardless of the detailed nature of the link, every link must have a defined interface in order to ensure that data is passed to and from its internal functionality in a predictable and correct fashion. This link interface is similar to subsystem public information description with respect to data requirements and outputs but, unlike the subsystem interface, includes information regarding activation flags and other context parameters which govern the dynamic treatment of the link and may vary with multiple instantiations of the link.

In keeping with the digital circuit analogy, links are represented graphically as block components on the plugboard. Data flows into and out of the link via terminals around the edge of the block. By convention, these terminals are sorted by function. Terminals on the left side of a link are used for setting input variables. Terminals on the right side of the link are the final output values of the link. Properties are a special case and are either explicitly connected via terminals or the top of a link if the link is a contained subsystem or referenced implicitly if the link is subject to an external constraint subsystem. A label beneath each link indicates the name of the activation flag that must be true in order for the link to be eligible for evaluation. In the case of permanent, continuous constraints, the activation flag name is replaced by the keyword ALWAYS. Examples of this convention are shown in Figure 4-3.

![Figure 4-3: The graphical interface specification of a link element](image)
In addition to specifying the activation flag of a link explicitly, the programmer has the option
to replace an individual flag with the keyword VIRTUAL. In such a case, the terminal of the
link may be connected to other plugboard elements just as any other link but the link itself does
not actually exist within the run time environment. This allows a virtual link to serve as a
graphical placeholder within the diagram for actual links which will be created at run time and
will inherit the local topology of the virtual link. The specification of a particular activation flag
for the actual link is done through the programming interface used to create the link. This
feature is useful for creating multiple instantiations of a particular link which all require the
same set of interconnections to the balance of the data flow graph and may be used both for
dynamic object creations and deletion as well as notational simplification.

The actual representation of the block representing the link is left to the designer. By default
the link's graphic simply labels the terminals for ease of wiring. Given the graphical nature of
the SHADOW UIMS, however, programmers may elect to replace the default labeling scheme
with a graphical icon more indicative of the underlying function of the link. Such an example
is shown in Figure 4-4.

![Diagram of link elements](image)

**Figure 4-4:** Alternate icons for a link element which returns the minimum of four input values

---

3 This function is part of the SHADOW link designer's toolkit available for use within the procedural body of
all link classes and is described in Section VI: User Interface Development Under The SHADOW System.
Links within the plugboard fall into two basic categories of implementation. They may either be the instantiation of a contained subsystem or a local constraint object. The former class of link is useful for developing complex subsystem from more rudimentary ones. In this case the public interface description for the contained subsystem defines and limits the ways in which the link may be wired to the other elements of the plugboard and all other details of the contained subsystem's implementation are hidden and isolated. This approach encourages code reuse, allows for the establishment of toolkits and library modules and helps to lower the overall complexity of subsystem implementations.

Encapsulating existing subsystems is only beneficial when subsystems exist that achieve the desired goal. Inevitably, new functionality must be provided. This need is satisfied by the second manifestation of links, embedded constraint objects. In this incarnation, a link consists of either a wholly encapsulated body of C++ compatible code or an interface to an underlying dialog independent application or external process. While the textual code within this class of link is C++ compatible, it is important to note that actual embedded code is only a collection of code fragments and that it is the responsibility of the SHADOW UIMS to manage issues of data scoping and parameter passing in a manner consistent with the SHADOW Talk paradigm.

From a functional standpoint, all links, regardless of specific class, have the potential to perform four types of computation within the SHADOW System. First, as mentioned above, a link may served as an interface to external C++ methods and/or other processes being executed concurrently by the underlying operating system or across a network. Second, a link may alter the values of any variables within its scope. This includes all local storage, link output
variables, and any properties defined within the associated subsystem. Third, a link may broadcast an event signal to the subsystem’s associated transition network. This may initiate state transitions and indirectly reconfigure the topology of the data flow network. Lastly, a link may raise an event signal to its containing supersystem. This function is useful for propagating events between subsystems.

**VARIABLE ELEMENTS**

The data flow graphs of a subsystem’s plugboard are composed of both links and variables. Links represent data consumers and producers and may or may not be present within the active topology of the graph at any given point in time depending on the state of their activation flags. Variables, however, are invariant data flow graph elements and serve as both continuous data conduits and short term data repositories.

There are six general classes of variables available within the SHADOW Talk UIDL, each with its own purpose and scope. A table of these classes, their scope, purpose, and graphical representation is given in Figure 4-5.

Unlike link elements, whose scope is strictly limited to the plugboard functionality, variable elements have the potential for much broader scopes. Variables may be visible in both the subsystem’s plugboard and its augmented transition network as well as forming the primary interface between the subsystem and the remainder of the user interface.

---

4These concepts are explored in greater detail in Section V: The SHADOW Run Time Engine.
### Class Table

<table>
<thead>
<tr>
<th>CLASS</th>
<th>SCOPE</th>
<th>GRAPHIC</th>
<th>PURPOSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Set in supersystem. Read in Subsystem</td>
<td><img src="image" alt="Name" /></td>
<td>Providing typed data to both the subsystem's plugboard and transition network from the container supersystem</td>
</tr>
<tr>
<td>Output</td>
<td>Set in subsystem. Read in supersystem.</td>
<td><img src="image" alt="Name" /></td>
<td>Exporting resulting data from the subsystem's plugboard to the container supersystem</td>
</tr>
<tr>
<td>Property</td>
<td>Potential global Shared by supersystem and subsystem</td>
<td><img src="image" alt="Name" /></td>
<td>Interfacing subsystem internals to both its local container and the global constraint engine</td>
</tr>
<tr>
<td>Explicit</td>
<td>Local to subsystem</td>
<td><img src="image" alt="Name" /></td>
<td>Exporting intermediate plugboard data to the subsystem's augmented transition network</td>
</tr>
<tr>
<td>Semantic</td>
<td>Used in subsystem. Published to shared memory</td>
<td><img src="image" alt="Name" /></td>
<td>Exporting intermediate plugboard data to other processes via shared memory accessed externally in a read-only mode</td>
</tr>
<tr>
<td>Implicit</td>
<td>Used by plugboard only</td>
<td></td>
<td>Visual indications of data flow between links elements.</td>
</tr>
</tbody>
</table>

**Figure 4-5:** The available variable classes and their graphical representations

Input variables, or stimuli, are specified as part of the subsystem public interface and may be referenced in a read only mode in both the plugboard and transition network. These variables are intended to be the primary conduit for information to be passed from a supersystem to its contained subsystem when encapsulating processing into modular structures.
Similar to stimuli, output variables, or responses are also specified at the public interface level and form the primary conduit for passing finished results out of the subsystem to its containing supersystem. Unlike stimuli, however, responses have a very limited scope within the subsystem. They may be assigned a value within the plugboard but cannot be referenced in any other fashion by either the plugboard or the associated transition network.

In addition to the restrictive input and output variables, SHADOW Talk also supports a class of interface variables called properties. Appropriate use of properties forms a powerful interface for the manipulation of subsystems both directly and indirectly in that properties combine the utility of input and output variables with a mechanism for dynamic superpositional binding allowing generic constraint subsystems to be created and applied as needed within the user interface without incurring the overhead or limitations of requiring static connectivity graphs for every potential target of the constraint. With regard to explicit use, a property is a public general use variable which may be referenced throughout the subsystem as well as its containing supersystem. When used indirectly, subsystem properties may be queried and modified by external, generic constraint systems which have been designed to find target objects based on the the properties that an individual object claims to possess. Thus, for example, a global rule for modeling gravity could be structured to alter generic properties of Position and Velocity for every object within the user interface which had the property of Mass associated with it.

In light of the various sources for modification of property values and the varied nature of the data a property may embody, properties have the additional capability to broadcast event signals to their associated transition network each time their value is updated externally. If
such event generation is requested by the designer, the modification of a property has the potential to spark state transitions within the subsystem resulting in the reconfiguring of the plugboard's topology.

In addition to the public interface variable classes, there are three additional classes intended for internal use within the subsystem. The first of these is the explicit local variable. All such variables must have unique names within the scope of the plugboard and the transition network. The primary utility of this class of variable is to pass information from the plugboard to the transition network for use in testing conditional transitions between states. Explicit local variables have no visibility beyond the bounds of the subsystem.

Semantic variables are nearly identical to explicit local variables with one important exception. In addition to read-only visibility within the subsystem transition network, semantic variable are also exported (again in a read-only mode) to a shared memory interface of the SHADOW UIMS. This allows external processes, such as a dialogue independent underlying application, to monitor the UIMS behavior asynchronously.

The final class of variables available is the implicit local variable. Within the graphical paradigm of SHADOW Talk these variables are the 'wires' that direct the output flow of one link into the input terminals of another. Implicit variables are limited in scope to the subsystem's plugboard. In essence, they are typed, unnamed data conduits similar to the connection lines of a digital circuit wiring diagram.

All variables, regardless of purpose or scope classification, must have a defined type. Currently the SHADOW System supports eight base data types: short and long, signed
integers; single and double precision floating points; single characters; character strings; generic pointers; and a UIMS specific pointer class. The majority of these data type have been designed for compatibility with those available in the C and C++ programming languages and have been named accordingly. The various data types, and their purposes are presented in Figure 4-6.

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>ASCII character values, compatible with C++ char data type</td>
</tr>
<tr>
<td>double</td>
<td>Double precision floating point values, compatible with C++ double data type</td>
</tr>
<tr>
<td>float</td>
<td>Single precision floating point values, compatible with C++ float data type</td>
</tr>
<tr>
<td>int</td>
<td>Signed integer values, compatible with C++ int data type</td>
</tr>
<tr>
<td>long</td>
<td>Signed long integer values, compatible with C++ long data type</td>
</tr>
<tr>
<td>ptr</td>
<td>Generic memory address information, equivalent to the C language's void* declaration</td>
</tr>
<tr>
<td>SHADOWPtr</td>
<td>Specialized address information, used internally to point to SHADOW subsystems within the UIMS</td>
</tr>
<tr>
<td>string</td>
<td>Sequenced, null terminated, ASCII character values, not to be confused with the C language's char* declaration</td>
</tr>
</tbody>
</table>

Figure 4-6: The available variable data types and their intended purposes

The interface designer or programmer has the option of associating a separate default value with each plugboard variable (with the exception of implicit variables, for whom the concept of a default value is semantically meaningless). Default values are statically defined and lexically

---

5The SHADOW System has been designed to allow the base data types recognized by the compiler to be extended to allow more elaborate, custom, and/or esoteric data types. This topic is discussed in Section VI: User Interface Development Under The SHADOW System.
scoped. The value is applied to the data storage area exactly once at the time of the object's creation within the UIMS. Once a variable has been updated (from whatever source) its default value is forgotten for the remainder of the interface's execution.

**DATA FLOW NETWORK SPECIFICATION**

A complete data flow specification consists of a diagram depicting the flow of information between link elements within the plugboard. Variables (named or implicit) serve as the conduits which connect the terminals of each link element with one another. At this level within the SHADOW paradigm, programming becomes analogous to digital circuit design and prototyping. Links represent combination building blocks (akin to integrated circuit packages), named variables are conceptually similar to buffers and simple latches, and implicit variables are the conductive 'wire' which binds all the elements together and defines the higher level functionality of the subsystem.

In keeping with the wiring diagram analogy, the visualization of the SHADOW UIDL uses conventions from electrical engineering to describe fan-out distribution of a variable's data within the plugboard. When a single variable is used as the input to several links, the intersection of the branching connection lines is marked with a filled circle (sometimes referred to as a meatball). This is done to differentiate intentional branchings from accidental line crossings resulting from a lack of planarity of the flow diagram. Where accidental crossings do occur, they should cross cleanly and, preferably, orthogonally. Connection lines should never be coincident. Figure 4-7 illustrates these rendering conventions.
Figure 4-7: Connectivity visualization conventions for data flow

By convention, plugboard diagrams are read from left to right, beginning with the stimuli and properties specified in the subsystem's public interface specification (as well as any local input links) and ending with the defined response variables and any local output links. An example of a complete specification is given in Figure 4-8.

In this example, the function of the subsystem is to control the behavior and rendering of a slider box within a defined slider trough geometry. Sliders are useful within WIMP style interfaces for indicating a particular value within a bounded range of values and are frequently incorporated into scrollbars and control panels. While the associated transition network for this subsystem is not presented here, it is still possible to trace the behavior and functionality of the plugboard by noting both the data flow lines and the activation flags indicated below each of the three links.
Figure 4-8: A plugboard specification of a simple slide bar

Two of the links hinge off of the activation flag ALWAYS, which indicates a permanent constraint within the subsystem. The right-most link is an output link and uses geometry information from the subsystem's properties and a local variable HandleX (indicating the slider handle's position within the trough) to draw the actual slider icon on the user's display screen.

The middle link is also a permanent constraint and is used to generate the HandleX variable needed by the output link. This handle position constraint object is responsible for calculating a position within the slider trough for the slider itself which accurately reflects the ratio of the current underlying value to the overall numeric range.
One of the key inputs for the middle link is the property Value. This variable is arguably the most important piece of data in the entire subsystem in that the subsystem itself is designed around presenting its visualization and providing a graphical means to edit its contents. As a property, Value may be both set and referenced by SHADOW elements external to this subsystem. Normally (as indicated by the uses of the ALWAYS activation flag) Value is only read internally and its contents are used to drive the generation of the visualization. The left-most link, however, provides a facility for altering the contents of Value internally in response to user actions.

The dragging constraint object becomes active in response to the activation flag DRAG being set to true (this is done by the transition network, discussed below). The activation of this link alters the topology of the network and allows mouse position data and slider range information to be used in the overall processing scheme. Once dragging has been initiated, the underlying value indicated by the slider becomes a temporary function of the X position of the mouse (the slider in this case having a horizontal orientation). This binding terminates when the activation flag DRAG is reset, presumably in response to the user releasing the mouse button. In this fashion, the other two links take on the responsibility of visualizing the current contents of Value without any need for understanding the origins of the data and write access to said data may be freely switched on and off in response to discrete events.

**EVENT MANAGEMENT**

In addition to enforcing continuous relationships between user interface objects SHADOW subsystems must be capable of reacting to discrete event occurrences. Within the SHADOW
Talk paradigm, this need is serviced by an event driven transition network. This network is, in
essence a simple state machine consisting of one or more states and zero or more transitions.

Under this system, a state is defined to be a unique set of link activation flags and their current
values. These activations flags are Boolean variables which are used to manage the topology
of the data flow network by selectively activating and deactivating individual links or groups of
links. Multiple states within a subsystem may share the same set of activation flags by name,
but the actual assignment of values to each of those flags should be unique to every state.

The SHADOW System does not require that every state explicitly specify the values for every
activation flag defined within the subsystem. Flags which do not change when transitioning
from one state to the next need not be listed. In addition, this feature may be exploited to create
the equivalent of Don't Care conditions with respect to unlisted flags when differing paths
through a network re-converge. When used properly, this practice may reduce the visual
complexity of some diagrams. When done improperly or unintentionally, such practices result
in non-deterministic automata which are difficult to debug and maintain. As with any
programming language, care should be taken to properly design and document each module
and the techniques used to realize it within the programming paradigm.

When creating a transition network specification, one state must be designated as the start state.
This state defines the default topology of the data flow network at the time of the subsystem's
initial activation (which, itself, is controlled by an activation flag in its containing
supersystem). Additionally, the programmer may specify whether the designated start state is a
simple default (to be used once and forgotten as the automata reacts to input events) or as a
general reinitialization condition to be recalled throughout the subsystem's lifetime. In the latter case, if the entire subsystem is deactivated and subsequently reactivated, the transition network will be reset to the designated start state; whereas, in the former, the reactivated system will simply pick up where it left off at the time of deactivation.

Consistent with other automata-based UIDs, a SHADOW subsystem moves from one state to the next by traversing transitions in response to input tokens. These tokens are named discrete event signals which have either been generated by links on the subsystem's associated plugboard, propagated down from the containing supersystem, or sent by the runtime UIMS engine itself in response to external modification of a monitored property.

The most common source for event tokens are user input devices themselves (accessed by the SHADOW System via input links). Unlike many user interface management systems, the SHADOW System makes no assumptions regarding the number of event classes available. This is done to allow for free extension of the base capabilities of the system as new and diverse input device technologies become available. Additionally, the SHADOW System supports event abstraction, wherein higher level subsystems may generate increasingly meaningful, context specific tokens in response to sequences and collection of lower level, more generic inputs. SHADOW Talk has been structured to allow the programmer to introduce as many event tokens as needed and imposes few restrictions with respect to the naming of said tokens. However, to promote code reuse and standardization within a development organization, strict naming conventions which reflect the current prevailing input techniques should be developed and followed when exploiting this language feature.
In addition to the activation token, each transition may also impose an additional restriction to determine if any given state transition should be allowed to occur. Such conditionals take the form of Boolean functions which may reference the current values of any named data areas visible within the associated plugboard via a parameter list. No attempt has been made to predict or limit the breadth or nature of the conditionals. Instead, the programmer is free to supply any C++ integer function with a visible, declared prototype within the scope of the subsystem. By convention, the SHADOW System assumes that any non-zero value returned from a user supplied condition function indicates that the transition should be allowed to occur, while a return value of zero indicates that the transition should be disallowed. The functions themselves are linked to the SHADOW Talk subsystem during final code generation.

**TRANSITION NETWORK SPECIFICATION**

The graphical representation of the transition network specification was designed to capitalize on familiarity with state transition diagrams commonly in use in many fields of science and engineering. A complete example of SHADOW Talk transition network is given in Figure 4-9. This example is the corresponding transition network for the slider subsystem whose data flow network was presented in Figure 4-8.

Within this structure, states are designated simply as circles with the default start state rendered with a double ring border. Associated with each state is a textual listing of its defining activation flags and their associated values. In the given example, the structure is quite simple,
there are only two states differentiated by the value of one activation flag. One defines the case
where the slider box is being actively relocated by the user and the other defines the case when
it is not. This latter case is the default situation at the time of subsystem activation.

\[\text{MouseDown}\]
\[\text{Inside}(\text{MouseX, MouseY, Rectangle}(\text{HandleX, YPos, Height, Height}))\]

DRAG = F

MouseUp

DRAG = T

**Figure 4-9**: The available variable classes and their graphical representations

While not demonstrated in the given example, individual states may also be explicitly named.
State names themselves are of no consequence to the SHADOW System itself but form a useful
mechanism for embedding supplemental information into the state diagram when dealing with
larger, more complicated systems. This is especially true in the case where multiple condition
flags are associated with each state and the function of each state is not obvious from the
permutations of the flags' settings.

State transitions within SHADOW Talk are represented by arcs which interconnect the state
bubbles. An arrowhead on each arc is used to indicate the direction of the transition. At a
minimum, each transition must be labeled with an input token representing the discrete even
which has the potential to trigger the state transition. Additionally, the arc may be labeled with
the name of a Boolean function which must evaluate to true at the time of the associated token's
arrival before the transition will be allowed.
There are only two events of importance in the given example: MouseDown, the token raised by an input link elsewhere in the system in response to the user depressing a button on a mouse; and, MouseUp, a similar token raised when a depressed mouse button has been released. Each of these events is associated with one transition arc.

By default, the MouseDown event is the only event with which the subsystem is initially concerned. If this token is encountered, the system will perform a check to see if the mouse pointer, indicated by the input variables MouseX and MouseY, is within the bounding box for the slider handle, (as calculated by a function called Rectangle and based on the slider’s geometric properties). If this condition is also satisfied, the transition occurs and the subsystem enters the drag state.

Once in the drag state, the activation flag DRAG is set to true and the dragging constraint link of Figure 4-8 becomes eligible for recalculation. Additionally, the subsystem ceases monitoring MouseDown events (since there is no transition out of the current state which depends on this token) and begins listening for MouseUp tokens. In this case, there is no additional restriction imposed on the applicability of the MouseUp token to the current situation. If such a token is encountered, the subsystem immediately returns to the non-dragging state, setting the activation flag DRAG to false and removing the dragging constraint link from the active data flow network.
GENERIC CONSTRAINT MANAGEMENT

In many cases, explicitly wiring all of the possible interrelationships between the interaction objects of a complete system is both tedious and impractical, especially in the case of distributed virtual environments. To address this issue, the SHADOW System provides a mechanism for specifying constraints in a generic fashion which are applied, as needed, by the UIMS to specific instantiations of selected objects.

In this fashion, for example, a virtual baseball game might be coded as multiple highly isolated player agents, a passive ball object, and a set of global constraints that govern the behavior of the ball (as well as other passive objects) whenever it is not under the direct control of an active agent. Thus, when one player throws the ball to another, the situation is modeled by the SHADOW System just as it is it would be in a physics class. At the moment the ball leaves the pitcher's hand, it is given an initial velocity, direction and spin but is no longer under the pitcher's direct control. For the duration of its momentum, the ball becomes governed by the global constraints of the virtual world which may be as complex as needed to supply the illusion of reality within the environment. The operation of catching the ball is the exclusive function of the catcher subsystem and the object being caught and the designer of the catcher subsystem does not need to address the issue of where the object came from to complete the activity.

To a much less complex and impressive extent, this same technique may be applied to traditional WIMP style interactions to create generic rules for the behavior of objects that might be awkward or cumbersome if explicitly embedded into individual subsystem specifications.
Some such applications might include facilities for context sensitive help and system wide error reporting and recovery utilities.

The specification of a generic constraint consists of two parts. The first defines the applicability or scope of the rule while the second defines the rule itself in terms of a generic representation of the rule's target.

**CONSTRAINT SCOPING AND PROPERTY BINDING**

The scope of a generic constraint is governed by the properties defined in the public interface specification of the individual subsystems of the system and the key set of properties associated with rule itself. For example, a simple gravity model might be written to apply to all objects within the system which possess a property named Mass. As in the case of event tokens, no restrictions or assumptions have been placed on the variety of properties available to the user interface designer. While this approach maximizes flexibility and extensibility of the system it also reinforces the need for strong naming conventions and careful design to allow for code reuse and maintainability.

Each rule in the generic constraint library maintains a list of core properties which an object must possess in order for the behavior embodied by the rule to apply to the particular object. In addition, the rule may require that additional information be associated with the object in order for the rule to be properly applied to the object in question. This is achieved through the specification of associated properties.
An associated property is one that may or may not already exist within the name space of any given object. If the named property already exists then it is bound to the generic constraint and shared between internal and external processing constraints. If the named property does not exist, a new data area is created and bound to the object for use by the generic constraint library.

![Diagram](image)

**Figure 4-10:** Generic constraint property binding and association specification

An example of this specification scheme may be seen in Figure 4-10. This diagram shows the property binding and association specification for a simple gravity constraint. Under this specification, the gravity subsystem constraints are defined to apply to all objects within the system which have the core properties of Mass and YPos defined. In addition, if the objects in question have the properties of YVel and GTime, a pointer to these data areas will be provided to the generic constraint, otherwise the additional properties will be added to the object and initialized by the generic constraint system.
A consequence of the SHADOW System's management of associated properties is that it also provides for a mechanism to create macro-properties, reducing the level of detail required to specify individual interaction objects within the user interface. For example, in a simulation the material properties of various objects may be of paramount importance but exhaustively specifying (potentially redundant) data for each and every object rapidly becomes tedious and error prone. In this case the generic constraint system may be used as a macro expansion system and a centralized look up table of key values. Thus, a program designer might specify that each object has a property named Material and associate a given value with each item. A generic constraint can then be structured to take all objects with Material defined and associate other properties such as Density and Elasticity and, based on the value of Material, assign appropriate initial values to the new properties. Properties added in this fashion become publicly accessible by any subsystem explicitly referencing them by name. The generic constraint itself may be null, indicating that the expansion is to be done once, at system initialization.

In some cases, binding a generic constraint to a particular class of subsystems based solely on a subset of the properties of said subsystem may result in rules which are awkwardly scoped. This occurs in two common situations. First, a generic constraint may be innately applicable to multiple objects simultaneously and dividing its application to each object individually may be counter-intuitive. This is the case when attempting to model a two object collision. In such an example, the rule is more correctly scoped to a pair of objects to allow the reactive behaviors of each object to be calculated simultaneously.
A generic constraint may be structured to address this issue of scoping to objects in pairs but such a specification leads to the second class of problem, over-extending the scope of a rule. A generic constraint which manages the interactions of two objects at the moment of collision should only be applied to objects which have collided. To accomplish this level of secondary discrimination, the SHADOW System provides a facility to specify a filter function when defining the scope of a generic constraint. This function is a C++ subroutine which must evaluate to TRUE for the given object or set of objects before a generic rule may be applied.

As illustrated in Figure 4-10, the existence of a filter function is completely optional. If no such function is defined, the generic constraint will be statically bound to all eligible subsystems at the time of the subsystems' creation and remain in effect over duration of the subsystems' existence. If a filter function is defined, then the binding becomes more dynamic. Objects which possess the core properties to satisfy the unconditional binding will have any associated properties added to their property lists at the time of object creation and will be considered candidate objects to which the filter function may be applied at run time. Each time the filter function accepts a given object (or group of objects) the generic constraint will be applied to the current data.

An example of the binding diagram for a two dimensional, two object collision generic constraint rule is given in Figure 4-11. The diagram indicates that the associated rule is applicable to any two objects with the defined properties of mass, position (XPos and YPos), and velocity (XVel and YVel) and that each unique pairing of such objects will refer to them as Body_1 and Body_2. Furthermore, for the rule to be applicable, the filter function Impact() must return TRUE for each instance of Body_1 and Body_2 identified.
Figure 4-11: Specification of a two object generic constraint property binding diagram with filter function

GENERIC CONSTRAINT SPECIFICATION

Once the applicability scope of a generic constraint has been established and any required associated properties have been bound to the targeted objects, the rule itself may be specified. This specification is structured in a fashion analogous to the subsystem plugboard and state automata diagramming techniques. Like ordinary subsystems, generic constraint systems may contain multiple embedded links or entire subsystems to perform intermediate processing and may use state transitions to reconfigure the topology of the data flow graph. Generic constraints, however, have no visibility into the internal workings or data areas associated with the specific target objects and may only effect said objects by means of defined properties.
Returning to the example of a gravity constraint, Figure 4-12 depicts a data flow graph of a simple constraint network corresponding to the binding properties of Figure 4-10. While it is legal for a generic constraint specification to have both a data flow graph and a state transition diagram, the given example is structured to have only one state and as such the state diagram has been omitted. In this system, a given target object is assumed to have the properties of a vertical position (YPos), a vertical velocity (YVel) and a time stamp (GTime) indicating when the effects of gravity were last applied to the object. (Recall that the property of Mass was also required for this constraint to apply to any given object but the mass information itself is not needed by this simple gravity model). The subsystem consists of three links: an input link which simply returns the current system time; a Delta T link which determines how much time has elapsed since the generic constraint was last applied to the object; and a gravity link which takes an initial position, initial velocity, and time step and returns the resulting position and velocity of the object, possibly setting these values to zero should the object come to rest on the ground\(^6\). Note that the object's properties serve as both inputs and outputs to the data flow network and are evaluated and updated from left to right.

\(^6\)Note that in this simple example no provision has been made to address bouncing objects, multi-object collisions or objects supported by immobile structures. A more detailed example of gravity and physics modeling may be found in Section VII: Evaluating The SHADOW Paradigm
**Figure 4-12:** Simple gravity constraint data flow specification

This method of specification allows constraints to be written in a generic fashion without concern for the specifics of the target objects' implementation or the number of objects so targeted. Additionally, the objects themselves need not be designed to be cognisant of the governing behavior of the world around them. Thus, in the design of a baseball simulation, the ball only needs to be aware of its position in space so that it may render itself on demand. Issues of how the ball attained its position and where it is going next, become the responsibility of the individual forces acting on the ball.

In addition to breaking complex constraint mechanisms, such as a sum of forces problem, in to a superimposed linear system of equations, the generic constraint system allows the user interface designer to maintain libraries of reusable generic constraints just as easily as he or she could for normal subsystems. Thus, time and effort needed to develop more complex behaviors and physics models may be amortized over multiple projects provided that all such projects adhere to the same property naming conventions.
SECTION V: THE SHADOW RUN-TIME ENVIRONMENT

OVERVIEW

The SHADOW Talk specification language carries with it certain assumptions with respect to how expressions in the language will be interpreted during program execution. While the implementation details of a run time engine built around the SHADOW Talk language are of little consequence (and would be subject to change from one platform to the next), a conceptual understanding of the underlying processing paradigm is essential to ensure a thorough exploration of the SHADOW System's applicability to the needs of non-WIMP style interface designers.

Many issues raised in Section III of this document directly relate to the topic of run time behavior and constitute the functional objectives of the execution engine working in concert with the SHADOW Talk language description. These criteria include: parallel (or concurrent) execution of conceptually concurrent activities; enforcement of continuous constraint relationships; automata-based constraint management; intermodule communication and synchronization management; and deadline driven autonomous time management.

At the highest level, a SHADOW program can be thought of as a dynamic set of constraints and dependencies combined with a set of rules which dictate the relative importance, ranging from required to forbidden, of each constraint at any given moment in time. The basic goal of the run time engine is to ensure that the most important constraints are satisfied and, as priorities change, so does the working set of important constraints.
Specifically, a SHADOW application is made up of multiple subsystems each consisting of a data flow graph and an augmented transition network. In the course of assembling the finished application the individual data flow networks are merged into a single, large graph which embodies all possible paths which data may follow within the confines of the SHADOW System. The individual transition networks are tabularized into rules which govern the topology of small localities within the master data flow graph and remain separate entities to be referenced only when an associated part of the data flow graph becomes an area of interest.

The first task of the run time engine is to review the overall structure of the data flow graph and to initialize its dynamic topology. This involves: categorizing and bookmarking certain types of links (INPUT, OUTPUT, and high priority, polled links) within the graph for future processing; scoping the applicability of generic constraints (possibly extending the geometry of the master data flow graph in the process); initializing all link activation flags based of the rules governing the individual automata's start state specifications; defining the topology of the working data flow graph based on the settings of the activation flags; and, marking any demand links associated with the initial activation flag settings as being eligible for calculation.

Once the system has been initialized, the run time engine has four basic goals. First, the engine must ensure that all links which generate user visible output are kept as up to date and accurate as possible within the bounds of real time deadlines. Second, the engine must react to any discrete events encountered in a timely fashion. Third, the engine must enforce continuous relationships in the face of changing data streams. Finally, the engine must make provisions to allow priority tasks to advance without compromising the other three objectives.
GENERAL PROCESSING PARADIGM

Unlike nearly every other programming language, SHADOW Talk was designed to support the philosophy that it is sometimes better to have a rough estimate of an answer in a timely fashion than to wait for a precise and accurate one. While this philosophy is not new (for example, an air traffic control system which can calculate the distance between airplanes in microns is of little use if it reports an answer of zero fifteen seconds after the planes themselves have collided), applications which require the ability to trade accuracy for speed have traditionally been outside the mainstream of computer science. As such, most languages do not provide innate support for time management or algorithmic contingency planning. Developers seeking to create time-sensitive applications are forced to either: unilaterally and permanently compromise the level of detail within the design to a level that current hardware platforms can keep up with; or, manually code provisions for time management at a level within the system where a high level language may become more of a burden than a blessing, and the long term maintainability of the code is at risk.

Within the field of software engineering, considerable time and effort has been expended investigating and evolving the tenets of modular structured programming. While this design and implementation approach has yielded many benefits to the computer science community as well as private industry, structured programming techniques are usually biased around functional decomposition of complex tasks. Once decomposed into modular units, functions and procedures are treated equally with respect to the run time behavior of the system. Like monochrome puzzle pieces in a toroidal jigsaw puzzle, all elements are perceived to make the same level of contribution to the completion of the whole. Issues such as intermodule control flow coupling and temporal decomposition are rarely addressed.
The SHADOW System was designed to provide support for the language features which non-WIMP user interfaces typically require but are outside the bounds of established language constructs with respect to mainstream programming languages and development tools. Within the SHADOW UIDL a designer or programmer may not only compose complex systems out of more simple ones, he or she is also given the ability to specify the relative importance of each subelement to the generation of the final solution. Additionally, the designer may specify computationally (and temporally) less expensive alternatives to particular subelements which the system may elect to substitute dynamically should time critical processing be at issue. Furthermore, the individual subelements, or links, are related to one another by virtue of the information content of the data that they consume and produce and have no direct control flow coupling.

Unlike third generation programming languages, the SHADOW System is largely declarative in nature and divorces the programmer from the concept of the call frame stack as the primary mechanism by which processing is passed from one routine to the next. Links within a SHADOW data flow graph do not call one another per se. Instead, a target link provides the run time engine with its data requirements and specifies other links which are capable of providing the needed information. At this level, the SHADOW run time engine has exclusive control over how control flow should proceed. The process of deciding whether or not a given link should be evaluated (or which of several alternatives should be selected for evaluation) is based on criteria specified by the programmer and reviewed with respect to the current run time situation.
Within the specification of a link's body of directives, the programmer retains direct control of the procedural flow of processing from one statement to the next. However, once the link is treated as a black box and embedded in a data flow graph of any non-trivial complexity, the specifics of how and when any given link is evaluated with respect to any other is beyond the programmer's ability to control or (in some cases) predict from a static specification. Links with no direct correlation, such as one might encounter when specifying parallel or orthogonal relationships, may be evaluated in any order which meets the run time system's temporal needs. Further, while links which depend on one another's outputs imply a sequence of evaluation, it is possible the run time system may chose not to evaluate a chain of dependencies in its entirety for reasons outside the scope of the individual links' specifications.

Restricting the programmer's direct access to the call frame stack makes the issues of effective temporal and control management of data flow graph on the part of the SHADOW UIMS of paramount importance. To address these issues and service the needs of SHADOW applications the user interface management system incorporates a run time component in the form of an executive monitor. This subsystem takes on the responsibilities of both solving and reconfiguring the data flow graph while maintaining real time data about its own performance for dynamic tuning within the bounds of programmer specified contingencies.

The SHADOW System was designed to model highly parallel, continuous behaviors, both on the part of the user and within the world. This conceptual approach has been incorporated into the design of the run time system itself. The basic processing model used in the engine's design is that of a system of highly isolated processing threads which may be executing sequentially, concurrently, or in parallel (depending on the hardware and operating system
capabilities of the target platform). Each thread maintains a public task queue into which other threads may place specific job requests as they hand off tasks to one another. If a given thread’s task queue is empty, the thread remains idle until a hand off occurs. Once a task has been placed in a queue, the associated thread will perform its task and remove the request(s) from queue on a first come first served basis as time cycles become available.

This conceptual architecture is reflected in Figure 5-1. The SHADOW run time engine consists of five core components operating under the umbrella of an executive monitor. One thread is tasked with the job of maintaining the dynamic topology of the data flow graph itself and services all requests for discrete event propagation as well as setting and clearing all activation flags within the master graph. The second thread the underlying constraint solver for localities within the data flow graph. It is the responsibility of this component, when given a target link, to assess the data needs of the link in question and to update as many links as is needed to service those needs without incurring unnecessary calculations or excessive processing delays. The third component is responsible for maintaining a database of links which have the potential to generate output directly visible to the user. From a display standpoint, these are the links which actually publish information to the user’s senses and as such occupy a key role in ensuring that deadline based processing is met. The fourth thread is also responsible for maintaining a database of links. These links, however, are input or internal processing nodes which the programmer has specified as being high priority elements which should be kept as up to date as possible even if there is no current demand for their output values (this can often be the case when passively observing a user or scanning a continuous data stream for a sequence that would constitute a discrete token in a given context). The final thread is responsible for managing the binding and application of generic (potentially global) constraints
and as such may be also be responsible for conceptual tasks such as keeping the elements of a virtual world up to date even if they are not currently being observed by the user. Each thread maintains its own task queue which dictates how it should behave the next time it is given a slice of processing time. The executive monitor addresses the issues of time management and the orderly handoff of tasks between threads.

![User Interface Specification](image)

**Figure 5-1:** The architecture of a SHADOW based application

Again, the view of the run time engine presented here is intended to convey the function and logical behavior of the SHADOW UIMS and makes no assumptions regarding the underlying platform on which the system has been implemented. This serves two purposes: the discussion of the run time engine's design may be addressed without introducing unneeded details which are subject to change from one hardware architecture to the next; and it reinforces an underlying tenet of the SHADOW paradigm that abstract specification should be buffered from implementation dependent complexities. From a programmer standpoint, disjoint graphs...
and generic constraints are inherently parallel operations. When executing on a single processor, run time system may interleave the evaluation of individual links to simulate the parallel nature of graph evaluations. Alternatively, in a networked computing system, the implementation of the UIMS may be such as to allow processing to be spread across multiple CPUs and multiple processes within those CPUs. In either case, task synchronization and process management is the responsibility of the execution engine working in concert with the operating system and need not be explicitly addressed by the interface designer.

THE RUN TIME EXECUTIVE MONITOR

The SHADOW Run Time Executive Monitor (RTEM) is responsible for managing processing resources between the six core component threads of the run time engine. This is the part of the SHADOW user interface management system which coordinates the tasks needed to ensure that the application's run time behavior is consistent with its SHADOW Talk description. In addition, the executive monitor accrues and analyzes performance data with respect to the run time engine's ability to keep the data flow graph up to date and adjusts the run time behavior model accordingly.

At a conceptual level, the six core component threads of the run time engine may be thought of as four perpetual cycles, a demand driven batch processor and a run time performance monitor. The three major processing cycles are dedicated to the automata maintenance, output generation, and the servicing of polled links. The fifth component is a constraint solving facility which serves as a utility subsystem for each of the four cycles. The final element
monitors timing data and assesses the systems ability to meet its deadlines. This breakdown is shown in Figure 5-2.

Figure 5-2: The architecture of the SHADOW run time engine

To discuss the interrelationship details of the executive monitor and the core components, it is beneficial to introduce some system specific definitions. A system iteration step is defined to be a finite amount of processing performed by the run time engine and is bounded in the temporal past by the end of the previous iteration step, and in the present by some discrete external deadline. For example, a given graphics system may accept frames for display every tenth of a second and the nature of a given application written for this system requires that each and every frame contain up to date information. This means that the system iteration step for
such an application would be the amount of processing done between the time when the previous frame was released for publication and the deadline for generating the next frame in the sequence.

With respect to the four perpetual processes mentioned above, a cycle is said to be complete if every item on the corresponding thread’s task list has been addressed at least once since the completion of the last system iteration step. Similarly, a system iteration step is said to be complete if each of the four processing threads was afforded enough time to complete a full cycle within the time frame of the iteration step.

Ideally, the executive monitor would like to ensure that all system iteration steps are complete, that is to say, all requests for processing time are accommodated at all times without missing deadlines. Unfortunately, ideal behavior is not always possible and in such cases, the executive monitor must take steps to reduce the workload allocated to each iteration step until it fits within the allotted processing time. The monitor has several potential mechanisms at its disposal to achieve this goal.

First, the monitor may elect to limit the amount of processing time devoted to one or more of the core component threads. For example, rather than trying to complete a full cycle of polled link processing, the monitor may limit the thread to serving only 75% of its outstanding requests within the given iteration step. Since the task lists are cyclic in nature, the thread can compensate for an incomplete cycle by resuming its processing with the next item on the list when it is given additional time in the future.
Second, the complexity of the tasks to be preformed may be reduced by applying faster, less accurate approximations. These algorithmic alternatives would have to have been supplied to the system by the programmer in the original specification. The ability to make such dynamic substitutions in the control and data flow of the interface forms the backbone of the SHADOW System's facility for supporting decimation.

Finally, depending on the architecture, resources and capabilities of the host platform, the executive monitor may have the option of allocating (or re-allocating) system resources to speed processing along critical paths. For example, in a multiprocessing environment, the executive monitor is in the ideal position to decide if the benefits of distributed processing within a particular thread outweigh the overhead incurred in the process of dividing, distributing and recombining the subordinate tasks and their results.

For efficiency, the executive monitor is the component of the SHADOW UIMS which is most closely tied to the host platform. While a simplistic implementation of the monitor's activities should function on nearly any Von Neumann architecture machine, to capitalize on the processing benefits of distributed processing networks and parallel multiprocessor machines, the monitor's implementation would need to be customized. The level of sophistication required for the monitor will depend on the degree to which contiguous and or parallel processing is supported on the target platform. Depending on the level of operating system support for process management and distributed processing, the complexity of the implementation may range from a simple collection of system calls to an entire subsystem dedicated to time and process management, resembling a subset of an operating system in its own right.
THE AUTOMATA MAINTENANCE FACILITY

The automata maintenance facility is responsible for propagating event tokens throughout the user interface's subsystem containment hierarchy and for updating system state information. Additionally, should changes in state result in the activation or deactivation of links within a subsystem, the automata maintenance facility must update network topology information for the associated locality of the master data flow graph. These various tasks are preformed by a collection of execution threads which each perform some subset of the required processing. In accordance with the the paradigm of the executive monitor itself, the automata maintenance threads hand tasks off to one another using internal job scheduling tables. Figure 5-2 illustrates the various entities within this facility and their relationships to one another and to other aspects of the run time engine.

The automata maintenance facility receives processing requests in the form of event records. An event record may be generated either by the run time system itself or programmatically by a UIMS system call from within the body of a link. In the latter case (the more common of the two), an event record has the potential to be generated whenever the constraint engine re-evaluates the link. When this occurs, the record is placed in the facility's Dispatcher Queue to await further processing.

7Where operating system facilities are available to perform a particular task, the run time engine should be structured to avail itself of them. This lesson has been learned at the expense of millions of taxpayers dollars throughout the 1980's and 1990's when many implementations of the Ada programming language attempted to blur the line between embedded and non-embedded programming by refusing to utilize operating system services for their intended purposes.
Figure 5-3: The interrelationships of the automata maintenance entities

The event record itself consists of a named token identifying the class of event, a tag identifying the event's point of origin within the interface specification, the nature of the event's generation (either BROADCAST or RAISED), and a system time stamp which identifies when the discrete event occurred.

---

8A detailed discussion of the differences between broadcasting events and raising events may be found in Section VI: User Interface Development Under The SHADOW System.
The Dispatcher thread takes event records from the Dispatcher Queue and distributes them to the affected (or potentially affected) subsystems within the containment hierarchy based on the input event masks defined for each module. In addition to internal processing within the originating subsystem, if an event has been generated in BROADCAST mode, a copy of the event record may be passed to any other subsystems which have the same containing subsystem as the originating link. To be eligible to accept such an event, each of these sibling subsystems must maintain an entry for the specific token in their respective input event masks. Additionally, if the initial event is accepted, copies of the event record may also be propagated downward within each of those subsystems' hierarchies for so long as the contained subsystems have requested such notification.

The broaden the scope of events within a system a link may generate an event in RAISED mode. Within the scope of the originating subsystem this is identical to a BROADCAST event, however, a RAISED event will also result in a copy of the event record being passed to the containing subsystem's parent and any of its sibling subsystems who have been designed to accept it. As with the case of siblings accepting BROADCAST events, RAISED events also have the potential to propagate downward along other branches of the containment hierarchy. If any subsystem within the event's scope of distribution accepts the given token, the dispatcher will record the act by placing the module on the Transition Pending Queue.

The State Manager thread takes entries from the Transition Pending Queue and determines whether an event token in a given automata's event queue results in a state transition, and, if so, what subsystem wide link activation flags need to be updated. When activated, the State Manager thread removes the next job request from the Transition Pending Queue. This record contains a pointer to a state transition table and the associated input event queue.
The first test performed by the State Manager is to confirm that the given token in the event queue is of interest to the current state. It is quite possible that any given event might be of general interest to the subsystem (and is therefore accepted) but is not currently considered an important or possible event in the current run time context. For example, a subsystem’s input event mask may include an event indicating that it wishes to be notified of incoming keystrokes but the interface’s design requires that the user must first click on a field before typing. If the required precursor event has not yet occurred the keystroke event may not be of interest to the subsystem. If contextual applicability test fails, the event record is removed from the local queue and further processing on the current job request stops.

Assuming the given event is applicable to the current context, the State Manager checks to see if any additional conditions must be fulfilled before a transition can be allowed to occur. If the target transition has a conditional activation filter associated with it, the State Manager makes a request of the constraint engine to provide it with reasonably up-to-date values for any local variables referenced in the filter function’s calling parameters\textsuperscript{9}. Using these values, the filter function is invoked and its return value inspected. The filter function itself may be any C or C++ function declared within the subsystem’s scope which returns an integer value. Should the conditional filter function fail to return a non-zero value, the State Manager terminates processing on the current task and moves on to the next item in the Transition Pending Queue (if any).

If both tests are successfully satisfied (or the first has been satisfied and no condition filter has been specified), the State Manager updates the target automaton and sets the associated

\textsuperscript{9}This may result in multiple link evaluations on the part of the constraint solver before processing on this thread may continue.
activation flags to reflect the state transition. Additionally, any subsystems which have undergone a transition are entered into the Topology Maintenance Queue to ensure that the master data flow graph properly reflects the current configuration of the activation flags.

The Topology Manager thread takes entries from the Topology Maintenance Queue and alters the activation status of links associated with recently updated subsystems. When activated, the Topology Manager removes an entry from the queue and inspects each link defined within the scope of the given subsystem to ensure that its activation flag is currently set to true. If the given flag is either currently \textit{TRUE} or set to the reserved word \textit{ALWAYS} and the containing subsystem is active, then the link in question is also set to active. If the link's activation flag is currently \textit{FALSE} or its parent subsystem is inactive, then the link is disabled.

If the status of a particular link is changed and the link is itself a contained subsystem, then that subsystem is entered as a candidate for further investigation in the Topology Maintenance Queue. This is done to ensure that subordinate subsystems are activated or deactivated in unison, allowing entire branches of the containment hierarchy tree to be switched on and off with the setting of a single flag.

The final operation performed by this manager occurs when a DEMAND link is activated. This class of link is intended for use as a discrete, one-shot processing element (similar to a callback function in traditional WIMP style interfaces). Once activated, it should be evaluated exactly once and in relatively short order after its activation flag has been set. Should such a link be activated in the course of performing topology maintenance, the link in question is inserted to the Polled Link Evaluation Queue as the next element for evaluation within that component of the run time engine.
THE CONSTRAINT SOLVER FACILITY

The constraint solver facility within the run time engine forms the heart of the SHADOW UIMS. It is this facility which directly controls the interpretation and execution of the interface's data flow graph and constitutes the only mechanism by which programmer specified link elements can be updated and evaluated. The unique part that the constraint solver plays in both control and data flow throughout the system mandates that it fulfill the role of a fundamental utility structured to service the needs of the other run time engine core components.

At the most basic level, the constraint solver is tasked with the job of resolving the data relationships between the interface's inputs and its outputs within the bounds dictated by the system's data flow graph. The graph itself is an elaborate system of structured elements, each of which may be classified as either a link or a variable. An abstraction of these structures is presented in Figure 5-4.

Both link and variable class elements maintain a dependency list. It is these lists which define the overall structure of the graph and this structure which defines the operation of the constraint solver itself. As the data flow graphs of individual subsystems are built, each element on the data flow chain refers back to earlier elements whose outputs are needed for the given element to function properly. This backward chaining continues until a system of data flow threads are defined for each output variable. This collection of threads allows the value of each variable to be traced back to the subsystem's inputs, uniquely defining which constraints must
be applied (and in what order) to a given set of inputs to perform the desired transformation from input to output.

![Link Structure and Variable Structure Diagram]

**Figure 5-4:** Abstractions of data flow graph elements

As larger, more complex subsystems are built from simpler ones, entire data flow graphs are concatenated together using this same backwards chaining technique. This allows the dependency lists to form loosely coupled modular structures where each subsystem is bound to the subsystems which it is cognizant of, and dependent on, and is responsible only for producing its intended outputs. Knowledge of how, where, or by whom such outputs will eventually be used is not required by the subordinate modules, reducing forward chaining coupling.

Conceptually, a SHADOW link element represents a one-way constraint relationship between its input and its output, one link in the chain of a greater transformation. From the standpoint
of the UIMS run time engine, the link represents an atomic unit of processing. Internally, each unit defines: a semantically clustered collection of sequential directives (the execution body); a set of preconditions which dictate when, if at all, the directives should be followed; the link's list of data dependencies; and a repository of output values based on its most recent update.

The SHADOW variable element is much simpler by comparison, having only one output and no internal processing capabilities. Furthermore, variable elements do not have any activation preconditions associated with them. They are considered to be active at all times and as such are permanent fixtures of the master data flow graph.

The data flow graph formed by the cascading of link and variable elements constitutes a superset of all data flow graphs possible within the system during run time. As discussed in the proceeding sections, all link elements have activation flags associated with them which allow the topology of the data flow graph to be changed dynamically in response to discrete event tokens. An illustration of this behavior is shown in Figure 5-5.

In Figure 5-5a, a sample data flow graph has been specified. This graph consists of seven links and four variables. Two of the seven links have conditional activation flags while the other five are keyed off of the reserved flag ALWAYS. This graph represents both the static specification of the master data flow graph as well as the dynamic graph produced when both FLAG:0 and FLAG:1 are set to TRUE.

Figure 5-5b shows the dynamic data flow graph produced with respect to link L:6 when FLAG:0 has been set to FALSE and FLAG:1 to TRUE. Note that the removal of link L:3
from the graph also eliminated variable \( V:0 \) as it is no longer directly or indirectly reachable from \( L:6 \). In contrast, link \( L:0 \), which is also an input for \( L:3 \) is not removed in this case since link \( L:5 \) remains active and also requires \( L:0 \)'s output.

Figure 5-5: A master data flow graph and its potential subgraphs

Figure 5-5c depicts the data flow graph which will result for \( L:6 \) if \( FLAG:0 \) is \( TRUE \) and \( FLAG:1 \) is set to \( FALSE \). In this case, the removal of link \( L:4 \) also eliminates links \( L:1 \) and \( L:2 \) from consideration despite their individual activation flags being set to \( ALWAYS \).

Finally, Figure 5-5d illustrates the resulting dynamic data flow graph that will result with respect to \( L:6 \) if both \( FLAG:0 \) and \( FLAG:1 \) are set to \( FALSE \). This graph also represents
the absolute minimum subset of the master data flow graph that contributes to the generation of $L$'s outputs at any point in time.

As the above discussion suggested, the view of the dynamic data flow graph is always taken with respect to a given element and traced backwards through active dependencies until independent elements (those with no entries on their dependency lists) are isolated. These nodes represent information sources or base inputs to the data flow graph. The path traversed to connect a given element to its base inputs defines the transformation that must be performed to generate the desired outputs for the given situation.

When one of the other core components of the run time engine requires a particular datum from the graph, a reference to the element is placed in the constraint solver facilities task queue along with return addressing information so that the calling process may be notified when the updated information is ready. The referenced element serves as the starting point for building a dynamic view of the data flow graph. For the constraint solver facility, the process of generating a given output reduces to a systematic exploration of each of the output element's dependencies followed by evaluation of the element's own directives (if any).

At first glance this approach may seem to be a simple application of functional programming, where the outputs of one or more functions become the inputs to others and run time behavior is characterized by the recursive exploration of subordinate functions. The fundamental differences between such a functional approach and the SHADOW paradigm lie in the SHADOW System's ability to selectively explore and evaluate element dependencies based on real time constraints and programmer specified criteria.
The goal of the constraint solver is not to produce the most accurate output (while this is certainly desirable and should definitely be pursued if one has the luxury of solvable, finite problems and sufficiently fast machines) the goal is to produce timely outputs which are as accurate as possible within the bounds of fixed deadlines. To achieve this goal the constraint solver facility always strives to do the bare minimum amount of work necessary at any given point in time and has the authority to prune both the depth and the breadth of a given element's dynamic dependency graph if timing is in jeopardy.

This pruning operation, however, is not done arbitrarily. The run time executive monitor provides the constraint solver facility with information regarding current timing and average overall performance to suggest to the facility just how many compromises need to be made. The programmer or designer may anticipate performance issues and specify links to be used for controlled decimation processing. Additionally, the constraint solver itself has built-in safeguards to prevent redundant evaluations of elements which lie in overlapping dependency chains to avoid wasting valuable processing time to begin with. Once the graph has been pruned and key link elements are identified, evaluation of the transformation from input to output may proceed.
Figure 5-6: The evaluation process of the constraint solver facility

A small example of this processing mechanism is given in Figure 5-6 (this corresponds to the data flow graph of Figure 5-5a.) The constraint solver is given an initial starting point for evaluation. In this case, it is the lightly shaded link on the extreme right of Figure 5-6a. From this starting point, the system traces the link's data dependencies back through the graph in a depth first search, marking each visited node as busy to prevent any cyclic references from
confounding the evaluation process. In Figure 5-6d the system has found a node with no further dependencies (in this case a variable). The node is evaluated (which for a variable means that its last known value is marked as current and available for processing) and the depth first search continues down the next closest dependency branch. In Figure 5-6f the system has identified a link with no further dependencies, like the variable node, it will be evaluated and its output marked as current.

By the time the system as reached the state depicted in Figure 5-6g, one branch of the original link’s dependency tree has been fully explored and a base of current input data has been identified. This allows the nodes along this branch to be evaluated. Once this has been done, the system is free to explore the next branch of the original link’s dependencies. Recursive exploration of this branch quickly finds that it is strongly coincident with the area of the tree which has already been explored and updated (Figure 5-6k). Recognizing this, the recursion stops and the single, unique node in this branch is updated.

This process of recursive exploration and evaluation continues until all of the data dependencies of the target link have been updated (Figures 5-6m through 5-6x). Finally, the target link itself is recalculated and the update task for the given link is complete and the constraint solver notifies the calling process that its requested data is available.

THE OUTPUT GENERATION FACILITY

As the name implies, the purpose of the output generation cycle is to produce the system’s output signals in a timely fashion. This is accomplished by maintaining a master task list, the Output Evaluation Queue, of all links capable of directly generating user detectable outputs
used throughout the entire user interface specification. Ideally, once per system iteration time step, each active item in the list is passed to the constraint solver facility, allowing each frame of output to be built element by element. The basic architecture of the facility is presented in Figure 5-7.

The output generation facility itself actually does very little other than list management. The designer or programmer is responsible for statically identifying output link elements to the system during interface specification by using the designation OUTPUT as the link's processing class. When an OUTPUT class link is instantiated (either at system initialization or as the result of a virtual allocation), a reference to the element is automatically placed in the Output Evaluation Queue by the output link task queue manager. Similarly, if an output link is deallocated during the course of the interfaces lifetime, the task queue manager will remove its reference from the evaluation queue to prevent further attempts at evaluation.

![Output Generation Facility Diagram](image_url)

**Figure 5-7:** Overview of the output generation facility
The enqueued link references serve as the starting point for re-evaluation of the application's outputs. At the start of each system iteration time step cycle the facility records the location of the current head of the task queue. From this starting point, the facility performs a quick check to see if the referenced link is currently active according to the dynamic data flow graph and therefore eligible for evaluation. If the link passes this test, its reference is handed off to the constraint solver facility to perform the actual update process.

Time permitting, the output generation facility moves on to the next item in the evaluation queue and performs a similar check and potential hand-off. This process continues until the end of the system iteration time step (as declared by the run time executive monitor) or until each node in the list has been explored exactly once.

If the facility was unable to complete a full generation cycle in the time allotted actions must be taken to minimize the likelihood of missing future deadlines. First, the run time executive monitor is notified of the failure so that it can adjust the dynamic load on the system accordingly. Additionally, the current position in the evaluation list is tagged so that the first link to be explored in the next cycle will be the first link left unexplored in the current step10.

THE POLLED LINK EVALUATION FACILITY

The Polled Link Evaluation Facility is similar to the Output Generation Facility in its basic function of dynamic re-evaluation of links and the data sources they depend on. It, too, uses a

10 This rolling evaluation is done to mitigate for any persistent timing problems which the executive monitor is unable to compensate for (such as a poor design specification or a grossly underpowered host platform). By varying the starting point for incomplete cycles and treating the list as a circular queue, every output link has the potential to eventually be explored.
master task queue to direct the scope of its work, and also makes heavy use of the Constraint Solver Facility to perform the actual updates. Figure 5-8 illustrates the strong architectural similarities to those of the Output Generation Facility (detailed in Figure 5-7).

![Polled Link Evaluation Facility Diagram](image)

**Figure 5-8:** Overview of the polled link evaluation facility

Unlike the output based evaluator, however, this cycle focuses on using entries in a Polled Link Evaluation Queue as the starting point for re-evaluation. This queue is designed for use by Input and Processing links which require either polling or discrete action regardless of immediate demand for their output values. In addition, the polled link evaluation facility is the mechanism by which all DEMAND links are processed and as such, is responsible for frequently inserting and removing references to newly activated demand links into its evaluation queue.
All dynamic links have some level of priority associated with them (a numeric value from 0 to 9 with 0 being the lowest priority). Normally links are assigned a priority of 0 indicating that the link should be evaluated only when its output is needed elsewhere in the system. If this is not sufficient for the task at hand (such as maintaining a history of eye movements or gestures for a syntactic event generator), a higher priority number may be assigned. This will result in the given link being classified as a polled link and references to it will placed in the Polled Link Evaluation Queue. The queue is managed as a circular list of link references with higher priority links being assigned multiple entries in the queue.

The queue is built by equally spacing redundant entries about a ring. The number of entries assigned to each link is a direct function of the priority number assigned (a priority of 4 results in 4 queue entries, etc.). Normally this queuing order is built statically and never altered during program execution, the one exception to this being the case of DEMAND processing links. When one of these links is activated (by the Topology Manager thread), it is immediately added to the queue as the next item to evaluate, allowing it to preempt the routine queuing order of the polled links. Once evaluated, however, the link is removed from queue and normal ordering resumes.

As in the case with output generation, at the start of each system iteration step the polled link evaluator records its starting point in the task list and begins exploring each node in the list to see if it should be updated. Unlike output generation, however, the polling facility will not necessarily stop once it has completed a cycle. In such a case, the facility will notify the executive monitor that it was able to perform a full round of processing (to help the monitor
better gauge overall performance) and will wait for authorization from the monitor to continue burning CPU time. If processing time is available (higher priority being given to maintaining current system output) the next available queue entry is re-evaluated and sent back to the end of the line to await its next turn.

When the executive monitor directs the polling facility to cease processing for the current iteration time step, the facility simply bookmarks its position in the task list for later continuation just as the output generation cycle does in the case of incomplete output cycles.

**THE GENERIC CONSTRAINT MANAGEMENT FACILITY**

The Generic Constraint Management Facility is responsible for scoping, binding and applying generic constraint subsystems to specific subsystem instantiations. Like the other update manager facilities, it maintains a dynamic task list and utilizes the constraint solver facility to perform the actual update. Unlike the other facilities, however, the entries in the Generic Constraint Queue cannot be passed directly to the constraint solver for evaluation, they must first be bound to specific elements of the master data flow graph. This mandates the introduction of an additional component, used to perform this binding prior to handing-off any update tasks. An overview of this facility and its components is shown in Figure 5-9.

While conceptually similar to the output and polled link facilities, the Generic Constraint Management Facility is the most complex of the three link update managers. Instead of maintaining a simple task list of generic constraints, the facility must record not only the constraints but also the applicability scope of each constraint with respect to the current composition of the entire interface containment hierarchy.
Figure 5-9: Partial overview of the generic constraint management facility

When the inclusion of a generic constraint is specified in the user interface, the system records the nature of the constraint, its scope within the containment hierarchy, and the list of core properties a subsystem must possess to be bound by the given constraint. Whenever a subsystem is instantiated within the scope of a generic constraint, the facility must test the new subsystem's property list against the binding list for the constraint. If the subsystem possesses all of the required core properties, it is added to the constraint's Target Subsystem List. Additionally, if the generic constraint specifies associated properties which the new subsystem does not already possess, the UIMS expands the subsystem's specification to accommodate them.

At the start of each system iteration step, the facility needs to record both its position in the master task list with respect to which constraint it is working on applying as well as its position
with the specified constraint's target subsystem list to identify which of the many possible targets is currently being addressed. In each step through the update cycle the facility tests an element from the current target subsystem list to see if its situation warrants updating. To be eligible, the link containing the referenced subsystem must be active and the generic constraint's filter function must return a non-zero value.

If a given subsystem is found to be a candidate for updating, the task initializer component is called to create temporary bindings between the generic data flow references found in the constraint's specification and actual variable elements specified in the subsystem's property list. Once this binding has been created, it is handed off to the constraint solver facility to perform a data update.

Like the polled link evaluation facility, this update process will continue to cycle for as long as it is allowed by the run time executive monitor. This can be especially useful when dealing with interface styles such as virtual reality where the model of the world should be continually updated whether or not the user is participating in (or indeed observing) the phenomenon.

**DYNAMIC LOAD MANAGEMENT**

The goal of the run time engine is to manage processing resources such that the UIMS may produce timely outputs that are reasonable approximations of the results that would be obtained if the system were not constrained by fixed deadlines. To achieve this goal the run time executive monitor strives to use every clock cycle to maximum efficiency while trying to minimize the overhead incurred by process management itself.
The dynamic load of the run time engine is a measure of the number of links elements (and their associated execution bodies) which must be evaluated within each system iteration step to properly service the immediate needs of the UIMS. This definition suggests three opportunities for reducing the dynamic load, all of which are exploited by the executive monitor. First, where multiple system outputs have coincident dependencies, redundant processing on those dependencies may be eliminated. Second, the dynamic load may be reduced indirectly by altering the set of immediate needs; this amounts to prioritizing tasks and isolating activities which may be temporarily deferred in the face of an encroaching deadline. Finally, the executive monitor may resort to decimation processing, where the depth and breadth of individual data dependency trees may be pruned in accordance with a decimation contingency specification and entire data transformation paths may be dynamically swapped out in favor of less computationally demanding ones.

**Figure 5-10:** A data flow graph with a strong potential for redundant processing
The primary mechanism used by the run time engine to manage the dynamic load is to avoid redundant calculations. Two situations where the potential for redundancy exists arise when conceptually parallel evaluations share part of a dependency tread, and when the updated output of a link element is needed and the link's own inputs have not changed.

Figure 5-10 illustrates the first of these scenarios. If the system requires that both links $L:6$ and $L:7$ be updated in the same iteration step, the possibility exists for links $L:1$, $L:2$, and $L:4$ to be evaluated twice since both $L:6$ and $L:7$ depend on them indirectly as inputs. Since $L:6$ and $L:7$ represent two distinct starting points for evaluation, the constraint solver facility has no direct way of detecting the overlap of the dependency threads. The run time engine compensates for this by introducing a system iteration step number tag into every link within the UIMS.

The step number tag records when the link was last updated with respect to the current iteration step. If the element is a target for evaluation and its tag number matches the current step number, the evaluation operation is skipped and a copy of its most recent output is used for future calculations. If the tag does not match, the link is re-evaluated, its output values are recorded and the tag is updated to inform any future exploration in this iteration step that the output data should be considered current.

A convenient by-product of this technique it that it also allows for synchronized updates of output channels when trying to simulate parallel activities on a serial processing platform. By

\[1^1\text{Note that this is a different problem than the one presented in Figure 4-6 where the exploration of dependencies from a single starting point crossed its own path.}\]
using iteration step numbers as a frame of reference, links with real time sensitivity such as a gravity constraint may be evaluated over many subsystems with respect to a fixed temporal frame. This allows the outputs of those elements to be presented to the user in unison as a coherent scene, independent of the actual order of evaluation internally.

The one exception to the iteration step redundancy check mechanism is the treatment of polled links. High priority links are allotted multiple entries in the polled link evaluation queue and as such should have the potential to be evaluated multiple times within the same iteration step. This situation is handled internally by differentiating the processing of the polled link itself versus that of the links which it depends upon (if any). The first time within an iteration step that a polled link is evaluated, its dependencies are explored just as they would be for any other element. Subsequent polling of the link, however, is strictly limited to the re-evaluation of the polled link's own execution body. While updating a link without updating its data dependencies may seem counterintuitive at first, it is important to remember that link polling is intended to be used for monitoring external devices. The execution bodies of most polled links should contain programmatic interfaces to other sources of information which lie outside the normal data flow domain and as such a polled link has the potential to change its outputs even if its inputs are static or undefined with respect to the dynamic data flow graph.

A second mechanism used to eliminate redundant processing is accomplished by caching the most recent set of input and output values for each link. With the exception of links dedicated to external I/O, properly structured links can be treated as black boxes which perform a deterministic transformation between input stimuli and output responses. The run time system exploits this behavior by assuming that any PROCESSING class link whose inputs have not
changed since its update, will continue to output its current values. When such a case is
detected, the system skips the evaluation of the link and simply updates the iteration step tag to
reflect the current time. Processing continues using the cached output values as if the link had
actually been evaluated.

Depending on the nature of the interface being built, the processing functions of the UIMS may
have varying importance. This idea lends itself to the concept of managing dynamic load by
deferring less important (or less time sensitive) activities to iteration steps that are less
computationally overbooked. For example, in the case of immersive virtual reality, the user
can be subjected to physical discomfort if the interface fails to synchronize the visual display
with the motion of the user's inner ear.

To address these concerns, the run time engine prioritizes the nature of the tasks that will be
addressed within each iteration time step. Whenever possible, the system strives to complete
the output generation cycle to ensure that the user is always given current feedback. The
automata maintenance cycle is also a high priority task in that it is capable of triggering both
system feedback and alterations to the dynamic data flow of the interface. Demand links
activated in response to discrete inputs should be serviced promptly, but may not be urgent
tasks with respect to the frame-by-frame behavior of the system. Finally, execution of
complete polled link and generic constraint evaluation cycles tend to be less important goals
with respect to any single output frame and are more flexible with respect to short term, global
deadlines. As such, individual polling and generic constraint evaluation tasks become the most
likely candidates for processing deferral from one iteration step to the next if time is running
short.
Recognizing performance problems is the responsibility of the run time executive monitor. The monitor continually gathers statistics regarding the ability of the UIMS to meet its deadlines and amount of processing actually performed within each iteration step. Once every second this information is reviewed and summarized in a system status variable known as the run level. This flag is a numeric value from zero to nine which indicates the severity of timing issues and the level cropping that must be done to the dynamic load to maintain a continuous output stream.

Normally, the run level is set to zero, indicating that timing is not an issue and all processing threads should be allocated sufficient time to complete their respective cycles. As processing continues, timing data for the actual performance of the system is accrued, the run time executive compares the number of iteration steps actually taken to a set of thresholds provided as part of the interface specification to determine if and how the run level should be adjusted. If the step rate is failing to meet a minimum acceptable threshold, indicating that the dynamic load is too great and must be reduced, the run level is increased. If the step rate is significantly exceeding the desired performance rate, indicating that there is processing time available to improve the precision and accuracy of results, the run level is decreased to signal that it is acceptable to increase the dynamic load of the system.

While the interface designer does not have direct control over the timing or frequency of any given link's activation, he or she is allowed access to the run level information and may use it as a means to structure controlled decimation of the interface's processing structure. Decimation is the process by which the computational expense of an operation may be reduced by replacing an elaborate algorithm with a faster but (for some context specific reason) less desirable one.
In addition to its normal activation flag, every link in the data flow graph has a run level window associated with it. For a link to be eligible for evaluation, the run time engine requires that the current run level fall within the window specified by the designer. If the current run level is too high or too low, the link (and all of its dependencies) are ignored. This allows the run time engine to prune both the breadth and depth of data dependency trees for any given link based on current performance data.

Additionally, a programmer implementing a practical application from an idealized design may exploit this feature and use the run level window specification as a selection mechanism to activate one of several parallel, algorithmic alternatives within a greater processing chain. This technique, while requiring more planning on the part of the designer and programmer, allows the run time executive to reduce the dynamic load of the system by replacing expensive calculations and rendering tasks with simpler, cheaper ones at any point in the chain from interface input to output. Similarly, if the run level drops for any reason, indicating that additional processing time is available, the substitution process is automatically reversed, increasing the dynamic load and producing more desirable results.\(^{12}\)

Between redundancy avoidance, prioritized processing and structured decimation, the SHADOW run time engine has the potential to maintain both a high level of user response time and continuous output streams. However, the varied nature and complexities of non-WIMP style interactions make it impossible to automatically guarantee performance under every situation. The SHADOW programming paradigm and the mechanisms embedded in the run time engine are simply the infrastructure around which the user interface designer may build an

\(^{12}\)A detailed discussion of planning for decimation and how to exploit the features of the SHADOW System may be found in Section VI: User Interface Development Under The SHADOW System.
optimum solution to address his or her specific needs. The SHADOW architecture was intended to provide a level of abstraction to simplify the process of transforming concepts into implementations, using the tool properly and efficiently is the responsibility of the designer. This issue is discussed in greater detail in the next section.
SECTION VI: USER INTERFACE DEVELOPMENT UNDER THE SHADOW SYSTEM

OVERVIEW

Within the SHADOW specification paradigm, the programming process consists of specifying a system of links and a governing logical structure which dictates how the links interrelate and when a particular link should be evaluated. The basic elements of the specification language consist of: augmented transition networks, which service discrete events and control data flow through the interface; customizable event tokens, which allow discrete signal passing between user interface elements and are used to initiate transitions within the automata; variables, which serve as data repositories and conduits during processing; links, which perform actual transformations of the internal data stream and serve as interfaces to entities outside the domain of the UIMS; subsystems, which are used to segregate, isolate, and modularize areas of the user interface specification into separable, reuseable entities; generic constraints, which are used to impose systems of data transformations over broad but selective areas of the internal data flow; and, a subset of the C++ programming language, used to specify the actual procedural processing embedded in each link definition.

Dual control (states and transitions) and data flow (links and variables) networks form the base expressive power of the SHADOW user interface description language while the constructs of systems, subsystems, and generic constraints provide the scoping and scaling facilities which
allow modular development. Using these constructs, a SHADOW system (or application) is said to consists of one or more subsystems and zero or more generic constraints. Both subsystems and generic constraints are, in turn, composed of local links, variables and augmented transitions networks and may wholly encapsulate zero or more other subsystems. This structure allows subsystems of greater complexity to be built from simpler entities forming a subsystem containment hierarchy. Within this hierarchy, multiple instances of a given subsystem specification may be included across the breadth of the tree but care should be taken to avoid encapsulations which would result in attempts at cyclical containment, invalidating the concept of a hierarchical tree structure.

The actual syntax of the SHADOW UIDL takes two forms, one diagrammatic and the other textual. The strong graphical nature of the language's entities lends itself readily to a visual programming environment consisting of a state diagram editor, a plugboard (or data flow graph) editor, a subsystem icon editor and a simple text editor for entering link execution bodies and embedded documentation. This graphical language has been dubbed SHADOW Talk and is described in detail in Section IV of this text.

The other form the user interface description language may take is exclusively textual and is lexically similar to SMGL. Being a pure text language, this variation of the specification dictates that the only development tool required to generate code is a text editor capable of saving files in an ASCII format. This language is referred to as SHADOW Script and is presented as an annotated grammar in Appendix A.

From the standpoint of the UIMS, SHADOW Talk and SHADOW Script are equivalent languages, in practice however, working directly with the textual language should be avoided.
SHADOW Script was intended to be the save file format for the SHADOW Talk editor portion of the UIMS and the input syntax for the SHADOW compiler. While direct programming in SHADOW Script incurs far less overhead, requires fewer tools and has the power to build very complex user interfaces, the syntactic structure of the language was not optimized for human interaction and programming may easily become tedious, convoluted and verbose.

In contrast, the use of graphical editing tools within the development environment is beneficial both for ease of specification as well as ease of maintenance. Studies done by commercial and government entities have shown that the use of accurate visualizations of legacy systems' architectures and implementations greatly reduces the overall costs and risks of long term system maintenance. While many applications are initially developed with the aid of visualizations such as flowcharts, entity relationship diagrams and calling hierarchy charts, maintenance of the specification charts in parallel with the evolution of the code base itself can be both costly and error prone. To realize the benefits of accurate visualizations, maintenance organizations will often resort to applying reverse or re-engineering tools to generate new graphs which reflect the current state of the code's evolution and can be dynamically queried.

By making the SHADOW UIDL inherently graphical these maintenance steps are simplified. Modifications to the visualizations of the application's structure are themselves modifications to the underlying code, ensuring that the visualizations are always synchronized with the current state of the code base. While no current programming language can make the claim to be fully self-documenting, the use of embedded diagrammatic specification in the SHADOW UIMS increases the likelihood that details of concept, implementation and intent will be far more apparent than they would be if a purely textual specification were employed.
Figure 6-1: The code translation process
The process by which a SHADOW interface specification becomes machine executable code is illustrated in Figure 6-1. First, the designer, working within a visual programming environment, produces a SHADOW Talk description of the desired functionality. The editing tools export this description in SHADOW Script format. At this point the specification is a combination of SHADOW Script tags and embedded C++ code fragments. This description is then fed to the SHADOW compiler which uses it to generate C++ prototypes and specification bodies. These files are then run through a standard C++ compiler and linked with library files, application object code and the elements of the UIMS run time engine to produce a stand-alone executable.

As this code translation scheme suggests, the SHADOW UIMS simply provides an additional layer of abstraction on top of the established domain of the C++ programming language. This provides the interface developer with a system which can draw upon the raw power of C++ when needed for specific tasks while divorcing him or her from many of the burdens of time management and other issues commonly encountered when dealing with the domain of non-WIMP style interactions. This section will explore some of the features and nuances of this abstraction layer from a developer's perspective and discuss techniques for exploiting these features for optimum effectiveness.

**SEQUENTIAL DEPENDENCIES AND PARALLEL PROCESSING**

To better service the needs of non-WIMP style interfaces, the SHADOW user interface management system provides language constructs and system services to allow an interface designer to create abstract specifications of parallel operations and concurrent processing while
divorcing the designer from the low-level details of actual control flow and process synchronization management. The cost of this abstraction layer is the ability of the programmer to directly manipulate and predict control flow within the application. Under the SHADOW paradigm, the designer may suggest guidelines for determining the flow of control throughout the system but may not dictate it. The actual control flow will be based on the system's immediate needs and available resources.

The SHADOW paradigm was designed to allow conceptual continuity between parallel and concurrent phenomena within the user interface and their specifications. This is accomplished by the use of the dynamic data flow graph in which transformations of the data stream are performed (at least conceptually) as concurrent activities on parallel branches of the graph. Unlike traditional procedural languages where explicit control flow instructions dictate a deterministic pattern of CPU resource application within a process, the SHADOW system uses data dependency threads and backward chaining constraint resolution techniques to determine the order in which links are evaluated.

Through use of the augmented transition network, a programmer may exercise limited power over the control flow of an application by preventing certain links from being evaluated and enabling others. It is important to note, however, that enabling a link is a distinctly separate operation from actually evaluating it. Similar to service requests under time sharing systems, the act of setting a link's activation flag to TRUE does not guarantee an immediate response. In such a situation, the run time engine records the link as eligible for evaluation but does not

---

13 In practice, the task of simulating true parallelism falls on the run time engine. Depending on the hardware and operating system services available on any given platform, the run time engine may be able to accomplish parallel processing by any one of several techniques. A further discussion of this topic can be found in Section V: The SHADOW Run Time Engine.
actually recalculate it until the link's output is needed elsewhere in the system. This procedure leads to two important points: the link activation order has no correlation to the actual order that any given set of links will be evaluated; and, the order of evaluation for multiple active links from disjoint areas of the dynamic data flow graph cannot be statically determined.

When the UIMS is alerted to the need for a particular action or data item, such as redrawing a screen element, the run time engine locates the associated link in the dynamic data flow graph and uses it as a starting point for a backward chaining exploration of the graph. As dependencies are identified and traced back to their initial inputs nodes along the chain are updated to ensure that when the evaluation propagates back to the initial node, it will be updated using the most recent data available. While the exact order of evaluation of the links is not predictable, this scheme does guarantee that in the absence of decimation processing (discussed below) each link in the chain will be evaluated after the links which it depends upon for input have been updated.

The concept of indeterminate evaluation order is illustrated by Figure 6-2. In this example, three potential orders of evaluation are presented. In each panel, the starting point of the evaluation is shown on the extreme right with its dependencies fanning out to the left. The number on each node indicates the link's position in the update sequence. While the exact ordering changes in each of the three alternatives, the basic property of any given node appearing later in the sequence than the nodes upon which it depends is preserved.
In most cases, use of the data flow driven order of evaluation is sufficient to service the needs of the application; the programmer's lack of direct control over sequential processing is not a problem. Situations may arise, however, where a particular sequence of link evaluations is required which is not directly discernible from the data flow itself. As an example, consider a WIMP style interface running on a platform with relatively simplistic graphics services available. Within this scheme there is a link which, when evaluated, will refresh the base window of the display and a second link, which when evaluated will draw a pop-up dialog box centered in the window. Each of these links has its own network of dependencies and both have the potential to be starting nodes for the run time engine's constraint solver. Such a situation is shown in Figure 6-3.
**Figure 6-3:** A trivial example of a system requiring a particular sequence of evaluation

If the interface associated with the schematic in Figure 6-3 were directed to refresh the screen while the dialog box were active, the run time engine would respond by updating both the window refresh node and the dialog refresh node. Unfortunately, the order of this updating sequence as presented cannot be determined. If the update of the dialog refresh node occurs before that of the window refresh, the dialog box will be briefly displayed and promptly overwritten when the base window is refreshed. Conversely, if the window is refreshed first, the scene will be updated and then a dialog box with be drawn over a subset of the window's area, exactly as one would expect. This results in a situation where two parallel paths with little data interdependence have an evaluation order which is both indeterminate and relevant.

While the example presented here is a trivialization of the problem, the same issue arises whenever two or more *OUTPUT* class links attempt to communicate with external devices over the same channel (in the above example the channel was the screen's memory map). Fortunately, the architecture of the SHADOW system provides for an easy mechanism to resolve such concepts. Links with little or no data interdependence within the bounds of the UIMS may have external couplings explicitly addressed by use of *dependency threads*. 
A dependency thread is a semantically specialized data dependency that allows links both to recommend an evaluation order as well as to pass signals regarding their own state of evaluation to other areas of the interface. Unlike an ordinary data dependency reference, the actual value of a dependency thread is not important and is not used within the execution body of the receiving link, only the thread's connectivity and its state (changed or constant) matter.

Figure 6-4 shows the same data flow graph depicted in Figure 6-3 but in this case a dependency thread has been added between the dialog refresh node and the window refresh node. Both nodes continue to be valid starting points for constraint resolution with the order of exploration of these starting points still beyond the direct control of the programmer but the existence of the dependency thread in conjunction with the architecture of the runtime engine ensure that the previous overwrite problem does not occur. In this case, if the dialog refresh node is explored first, the dependency thread forces a complete evaluation of the window refresh node before the dialog node can be updated. When the system initiates an evaluation at the window refresh node, rather than overwriting the output of the dialog node, the node reports that it has already been explored within the current system iteration step and processing terminates. If, on the other hand, processing begins with the window refresh node first, the window is redrawn and then the dialog refresh node is explored. The dependency thread simply reports that the referenced subtree is up to date (no further exploration or evaluation required) and processing continues with the dialog box being drawn over the existing window display. In either case, the existence of the dependency thread eliminates the problem of concurrent processes accidentally overwriting one another's output while incurring a minimal amount of processing overhead.
EVENT ABSTRACTION AND PROPAGATION

In addition to the task of processing continuous, parallel streams of data, the SHADOW System also needs to recognize and respond to discrete events which signal a change in the way overall processing should be performed. Unlike many traditional toolkits and management systems, the SHADOW System make no assumptions about the number or class of events to be encountered in the course of designing and specifying an interface. Any discrete signal may be generated in the form of a SHADOW event token. Tokens are identified by lexical names provided by the designer and may be generated, propagated, and serviced by name throughout the containment hierarchy of a finished application.

The scope of an event is dependent on the manner in which it is created. An event may be directed at an individual subsystem, designated for internal processing by the subsystem which generated it, or may be intended for both internal and external use, propagating both upward
and downward in the containment hierarchy tree. Regardless of scope, it should be remembered that event tokens are designed to allow entire subsystems to signal one another, not individual links. The target of an event is always the augmented transition network associated with a given subsystem.

The function of an event token is to create the potential for a state transition within one or more augmented transition networks. States are defined by their unique collection of link activation flag settings. Performing a state transition amounts to reconfiguring the dynamic data flow graph to alter the transformations used when converting input stimuli into output responses. Additionally, a state transition may result in requests for demand processing links to be serviced to perform discrete actions in response to discrete events (similar to callback functions under traditional WIMP toolkits).

Witnessing an event token does not, however, guarantee that any given system will perform a state transition. There are several conditions which must be satisfied before a given event becomes meaningful to a subsystem. First, the event must apply to the current state. If the augmented transition network is in a state that has no exit path associated with the given event, the event is ignored. Second, if a path exists, it may have a programmer specified conditional associated with it which restricts when the path may be exercised. For example, a mouse click event may be of no interest to a GUI screen button if the mouse pointer is not positioned over the button's real estate. Finally, if the event was generated externally, the subsystem may summarily reject the event based on its own event mask which restricts outside influence on the subsystem's internal processing.
Events are generated by two sources: the UIMS itself; and, programmer-designed links. The SHADOW run time engine has the potential to generate automatically a limited set of events under very specific conditions. As with all events, these tokens are identified by name and are subject to the same filtering process described above. The system event token set includes: the SHADOW_FrameComplete token, which is sent to the root of the application's containment hierarchy tree at the end of each system iteration time step; the DONE token, which is issued in response to a demand processing link service request being satisfied and is sent to the subsystem which requested the service; and, an assortment of property-value-changed events which are sent to subsystems which have indicated that they wish to actively monitor updates to a particular property variable.

While system generated events are useful, the primary source of event tokens are programmer defined links which generate events through UIMS system calls. This allows designers to create as many tokens as desired, to name the tokens in a fashion that is descriptive of both their application and their context, and to designate the scope of tokens as either an internal event, an event with both internal and external significance, or a directed message to an external subsystem.

All event tokens are identified by lexical name. While this provides great flexibility to the designer in creating new tokens, it also raises the possibility of accidental collisions in the token namespace as individual subsystems are brought together and integrated into a coherent, complex system. To reduce the likelihood of such collisions, each subsystem maintains an event mask which both declares the names of events that it has the potential to produce and distribute externally and presents a list of the event tokens which the system is willing to accept.
from an outside source\textsuperscript{14}. With the exception of system generated tokens, if a subsystem is exposed to an external event which is not explicitly named in the list of accepted events, the event is blocked and the subsystem's augmented transition network remains oblivious to the event's existence.

The scope of an event is a function of the manner of distribution requested at the event's point of origin, the architecture of the application's hierarchy containment tree, and the event masks specified for each subsystem within the tree. When an event is first generated, the originator requests one of three possible distribution patterns: BROADCAST distribution, where the event is used for internal processing and may be passed to its contained subsystems if they choose to accept it; RAISED distribution, where the event is used internally as with broadcast distribution but is also transmitted externally to the subsystem parent in the containment hierarchy; and POINT-TO-POINT transmission, where the signal is ignored internally and is sent to exactly one known location in the hierarchy for internal processing.

Figure 6-5 illustrates the various scopes resulting from the different distribution methods. In each case a visualization of the containment hierarchy tree is given with those subsystems who accept a given event shown as a white superellipse while those subsystems which do not accept the given event token are marked with a gray 'X'. The effective scope of the event for each situation is depicted using background shading. The subsystem which generated the event is shown at the top of the leftmost major subtree.

\textsuperscript{14}The term outside source in this case refers to the subsystem's immediate ancestor in the hierarchy containment tree. If the subsystem is built from other subsystems which are capable of raising events, these subsystems would be considered insider sources and are not subject to the filtering effects of the event mask.
Figure 6-5: The effective scope resulting from various event distribution methods.
In Figure 6-5a, the event has been generated in broadcast mode. This means that the event may be used internally by the originating subsystem's own augmented transition network, and that copies of the event record will also be propagated down the tree to any of its contained subsystems which have listed the given event in their event masks. In this case two subsystems accept the event, one of which goes on to propagate the event down to one of its own subsystems which also lists the token as an acceptable event.

In Figure 6-5b, raised distribution is requested. This is a superset of the scope defined for broadcast distribution. In addition to broadcasting the event internally, the originating subsystem also passes a copy of the event up the containment tree to its own parent who treats the event as if it had been generated internally in broadcast mode. Note that in this case there is a subordinate node on the extreme right which has indicated that it will accept the given event but that its container (which has the potential to witness the event) does not. This results in the event being blocked from propagation down this subtree, limiting the scope of distribution.

Figure 6-5c illustrates a point-to-point transmission from the originating subsystem to the leaf node of the right hand subtree. In this case, the given event is not used internally and the structure of the containment tree between the source and destination of the event is irrelevant. To use this form of distribution the originating subsystem must know the exact address of the target subsystem. All system generated events are sent as point-to-point transmissions.

In addition to providing a flexible mechanism for defining and scoping low-level discrete tokens, the SHADOW paradigm allows the designer to introduce layers of event abstractions. This serves two purposes. First, propagation and processing of low-level, related events may
be localized within the specification to reduce coupling between subsystems. Second, readability and traceability of the code may be improved by renaming events with labels that better reflect the context and semantic meaning of the event as processing propagates.

For example, a system might be built where one subsystem monitors the mouse pointer status and other subsystems manage a traditional WIMP style interface. A subsystem managing an on-screen help system listens for a generic MouseClick event from the mouse subsystem. When it witnesses such an event within its domain, the subsystem generates its own event with a more specific name such as HelpScreenRequested and propagates the context specific event in place of the generic one.

At a more sophisticated level, subsystems may absorb elaborate chains of low-level events and, upon recognizing a target subsequence, generate macro events to be processed at a higher level of the application. This is useful for localizing event processing within the specification and, when combined with the inherent parallel nature of the SHADOW paradigm, may also help to parse probabilistic tokens such as gestures and eye motions.

**DATA ABSTRACTION**

Data abstraction within the SHADOW paradigm takes several forms depending on one's perspective and one's needs. From one perspective, the construct of a SHADOW subsystem itself may be thought of as a form of data abstraction in that subsystems are similar to objects
within an object-oriented paradigm. They are used to bundle collections of base data elements together and to couple those elements with the methods which access and manipulate them while maintaining a rigid interface with the outside world. Alternatively, from a more traditional perspective, data abstraction may be thought of as the ability to define new data types and to pass structures and unions from one function to another. To a limited extent, the SHADOW system supports these operations as well. Within the body of a link's execution code, a programmer has direct access to any externally defined C++ data abstraction or class definition. Additionally, the SHADOW run time engine has been designed to allow (relatively) painless expansion of the set of native data types recognized by the UIMS.

When architecting an interface within the SHADOW paradigm, it is often reasonable to think of subsystems as objects or object classes. Each subsystem maintains a rigid data interface which defines public inputs, outputs, and properties of the entity while isolating internal operations and data from external processing. The internal processing may be as complex as required to achieve the subsystem's objectives, and operates in a private namespace to support modular design. Links act as object-oriented methods which update internal data and transform input stimuli into output responses. Links, themselves, may be the instantiations of other subsystems used to build more complicated processing streams from simpler building blocks. Moreover, multiple instances of such individual subsystems each have their own data area and are processed independently of one another.

The SHADOW user interface description language is not, however, an object-oriented language in the strictest sense of the term. Subsystems do not inherit from one another (as is commonly the practice in OOP). Instead, SHADOW subsystems use a containment paradigm,
where a derived entity will explicitly embed another subsystem within itself as a means of
drawing upon its functionality. In this way, the data transformations of a generic subsystem
may be customized by use of preprocessing and post-processing links within its containing
parent. Additionally, SHADOW Talk has no mechanism for implicit polymorphic behavior.
In all cases, a subsystem's data flow graph explicitly defines data stream transformation
process for all inputs within the scope of the module.

The proper use of SHADOW subsystems offers distinct benefits for the interface designer.
From a practical standpoint, they provide for modularization of data and data processing
methods and facilitate efforts at code/object reuse. Additionally, subsystems are conceptually
well suited to model real world entities, and can be used to support design methodologies
which model a virtual environment as a collection of black box constructions.

While applying SHADOW subsystems to the problem of modular design and construction
facilitates data abstraction and management internally, it does nothing for dealing with abstract
data types which need to interface with entities outside the UIMS. Such interactions can be
quite common when trying to integrate commercial off-the-shelf packages such as graphics
rendering engines or networking communications packets. To deal with such interactions, the
SHADOW system allows links to have direct access to all forms of C++ data abstractions.

Provided proper C++ header specification files are referenced, any user defined class, union,
structure or type definition may be used within a link's execution body just as they would be in
any C++ code fragment. Since links form the only direct interaction the UIMS has with the
outside world, this localized approach to handling external data abstractions can satisfy many
application specific needs. However, within the greater SHADOW paradigm, plugboard variables are strictly limited to the native data types defined within the run time engine for the UIMS. This does not mean that it is impossible to pass external data structures from one link to another, it simply makes process a less elegant and more error prone process.

While abstract data structures themselves cannot be directly passed from one link to another via the data flow graph, the memory addresses of such structures can. One of the native data types supported by the run time engine is a generic pointer (the equivalent of a void* in C or C++) which may serve as a conduit for passing addresses from one link to another. Under this scheme, it becomes the responsibility of the link receiving the address information to apply the correct type cast to recover proper access to the data within the structure. Failure to do so will result in unpredictable consequences.

The approach to data passing is further complicated by the optimizations of the run time engine with respect to link re-evaluation policies. An active link will only have its execution body evaluated if its output is needed and its input has changed. If data within a structure has changed but the address of the structure has not, then the run time engine will not notice that any link relying on this data is out of date. To correct for this, dependency threads may be introduced and coupled to particular data structures. Each link that modifies the contents of a data structure should also record this fact by sending a signal out onto the dependency thread to ensure that any link depending on the data within the structure will be updated appropriately when needed.

The use of generic pointers coupled with dependency threads can be undesirable for several reasons. First, it increases the responsibilities of each link with respect to the amount data
handling it must do to operate effectively. This increased level of responsibility also increases the risk of a link failing to fulfill its obligations properly. Such opportunities for error include: failure to recast a generic address back into a typed data structure; casting an address into an incorrect data structure; and, failure to designate when a data structure has been updated internally. Additionally, the use of type casting defeats data flow graph level type validation, and the use of an extra dependency thread for every address passed has the potential to seriously clutter data flow graph diagrams.

To address these concerns, the SHADOW system was designed to allow its set of native data types to be expanded. This is not a process to be undertaken on an application by application basis and is not done within the bounds of the SHADOW UIDS. Extending the native data types recognized by the UIMS requires the coding of additional run time engine elements in C++ and a rebuilding of the compiler itself. The procedure, however, is not nearly so painful as it sounds. Fabrication of the run time engine extension consists of three basic steps assuming the source code for the UIMS is available: creation of a subclass of generic plugboard variable class; overloading a specific set of assignment and comparison operators; and providing code which monitors changes to internal data elements and sets a data updated flag when appropriate. Actual integration of the new entity is a relatively minor task since the underlying system was designed for expansion. This technique was used to create the native string class provided by the UIMS.

The process of extending the base data types for the plugboard can be somewhat complicated but may well be worth the investment if the given data structure has a high probability of widespread use. By creating a data type which has an accurate data update flag, the use of
dependency threads coupled to data structures may be eliminated, along with the risk that a link will fail to mark the data as updated, and also making data flow graphs less cluttered. Additionally, the expanded set of native types allows the compiler to perform stricter type checking and to reduce the risk of erroneous type casts.

**FUNCTIONAL DECOMPOSITION**

The SHADOW system provides mechanisms to support modular design and implementation at both macro and micro levels. On a broad scale, interfaces may be broken down into hierarchies of loosely coupled subsystems. At a finer level, the functionality of a subsystem may be decomposed into individual links. The key to a successful, maintainable design, however lies not in the existence of these mechanisms but in their disciplined application by the interface designer and programmer.

The use of subsystems for modular decomposition of application-wide functionality is simply the SHADOWTalk implementation of a familiar concept and, as such, does not constitute a radical conceptual departure from conventional software engineering practice. In practical terms, this means that many of the concepts embodied by techniques and guidelines developed for large scale development efforts, such as entity relationship diagrams, data structured system design, etc. are equally applicable to the SHADOW paradigm at a macro level.

When dealing with the functional decomposition of the internals of an individual subsystem, however, the design of the run time engine becomes a driving consideration. Unlike functional
or procedural languages, the SHADOW specification language supports loosely coupled control flow and is optimized for meeting deadlines rather than for exhaustive processing. These factors must be considered when deciding how processing is to be divided among links.

From the standpoint of the run time engine, a link represents an atomic unit of processing. The UIMS has the option of evaluating it or not, as circumstances dictate but once the update process for a given link has begun, the run time engine has no provision for preempting it. This means that if a programmer attempts to put too much processing inside a single link, the link has the potential to become a processing bottleneck, resulting in missed deadlines on the part of the run time engine.

At the other extreme, decomposing a processing stream into dozens of trivial links not only clutters the data flow graph, it also may result in poor performance. Unlike procedural languages where a direct subroutine has a relatively flat, nominal expense associated with it, the loose coupling of links means that exploring a data dependency incurs far more overhead than simply pushing a call frame and setting the program counter. When trying to update a given value, the run time engine must visit each and every active link that potentially contributes to the value and must decide on a case by case basis whether or not to update the given dependency. If links are reduced to trivial processing complexities, it is possible for the UIMS to spend more time deciding which links to update than actually performing useful processing.

While the optimum size and complexity of the execution body of a link remains a topic of debate, experimental data suggests that less than ten lines (barring calls to CPU intensive
external subroutines) is too small; and that links with a cyclomatic complexity of five or more
could usually benefit from being further decomposed. It is also important to remember that
individual links may be selectively activated or deactivated, allowing conditional branches with
algorithms to serve as a criteria for decomposition and using the augmented transition network
to dictate which path the data should take at any given point in time.

Finally, functional decomposition should be structured to capitalize on the features of the run
time engine. The designer should recognize that run time consideration may preclude complete
evaluation cycles, and some links in a dependency chain may be skipped in order to meet
deadlines. The designer should take care to provide evaluation break points at reasonable
levels in the dependency tree to ensure that deadlines are met and a reasonable approximation of
the desired output stream is continuously maintained. Additionally, the loose control flow
coupling of links allows for the easy introduction of algorithmic contingency plans should
performance problems become an issue and creates an infrastructure for creating planning
decimation processing.

**PLANNING FOR DECIMATION**

Traditionally, decimation has been used in virtual reality systems to speed graphics rendering
problems. For example, the exterior of a virtual car might be ideally modelled as a three
dimensional network of polygons with surface shading when the viewer is sufficiently close to
warrant a high level of detail. This detail is wasted (as is the processor time that went into
producing it) if the viewer is so far removed from the car that the vehicle only occupied a few
hundred pixels in the scene. In such a case, a programmer may specify a simpler model of the
car to be used in situations where a high level of detail is not required or practical.

Figure 6-6: Implementing manual decimation within the SHADOW paradigm

The SHADOW paradigm readily supports this form of decimation through its network of links
and their associated activation flags. A given object in a virtual environment may have several
possible renderings specified for it, each embodied by a separate link with its own activation
flag. When conditions warrant selecting one output link over another, a link within the
subsystem raises an internal event and alters the activation flags accordingly. As the run-time
engine builds each output frame, all possible output links are checked but only the one marked
as active is actually evaluated. Figure 6-6 shows a simple implementation of such a
specification.
While effective, this form of decimation is something of a manual process in that it is the responsibility of the subsystem to select which of several possible end points the system should pursue; the selection process itself is based on a static assessment of the world rather than actual run time performance data. The SHADOW system recognizes these limitations and provides additional infrastructure to allow a designer to create contingency plans for automated decimation to be invoked by the run time engine as needed, based on its ability to meet its real-time deadlines.

The use of structured links in a data flow graph, combined with the run level status flag, allows the SHADOW system to take the basic concept of decimation and extend it throughout the entire scope of the interface specification. Every link has an activation window associated with it. This window defines both the minimum and maximum run levels the system must be at to allow the link to be re-evaluated even if its activation flag is set to TRUE or ALWAYS. By default, all links are created with a minimum run level of zero and a maximum of nine, indicating that their eligibility for recalculation is strictly a function of their activation flags. The designer may change these settings, however, allowing a contingency plan for automated decimation processing to be built.

Within the SHADOW paradigm planned, automated decimation takes two forms. An expensive processing activity may be either pruned, where only part of its calculations are performed, or replaced, where less expensive algorithms are substituted. Pruning amounts to using the run level windows of a set of links to abort dependency exploration beyond a certain point if time is an issue. Replacement is similar in concept to manual decimation described
above, but the link selection process is based on run level as well as activation flag settings, and is done by the run time engine rather than the subsystem.

As an example of pruning, consider a virtual world which includes an area where several icosahedra are rotating on their axes while bouncing around in a virtual chamber. For the purposes of this interface, the rotation of the shapes is done simply for aesthetics, but the position of the shapes in space must be rendered accurately at all times. A simple abstraction of such a system is shown in Figure 6-7. Within this scheme, the rendering of each individual icosahedron depends on its position in space and its current angle of rotation. According to the activation flags, all links should be considered active at all times but the links which manage the rotation data have the additional restriction of a narrow run level window. The Render Shape and Move Shape links use the default run level bounds of zero and nine, indicating that they should always be explored when needed. The Rotate Shape link has a maximum run level bounds of one, indicating that if the system becomes bogged down (run level increases), the link's contribution to overall data flow may be safely ignored.

Figure 6-7: A simple pruning example
According to the specification given in Figure 6-7, if the run time executive monitor raises the run level to two or higher, all rotation calculations on all active icosahedra will be ignored. This results in decreasing the dynamic load on the UIMS and frees up CPU time for performing the more important spacial position and rendering calculations. If run level subsequently drops back down below two, all rotation calculations will automatically resume.

A second form of automated decimation supported by the SHADOW specification paradigm is algorithmic replacement, where expensive individual steps in a larger algorithm may be swapped out and replaced by fast running approximations within the overall data stream. As in the case with pruning, replacement relies on the use of individual link's run level windows to determine which links should be used in the course of evaluation at any given time. Figure 6-8 shows a simplified\(^{15}\) abstraction of a dependency chain which makes use of this principle.

The processing example given in Figure 6-8 describes a system which needs to know the area under a curve and finds this value by sampling the curve within a set of boundaries and performing some form of numeric integration. The specification provided states that there are three ways the subsystem may perform the numeric integration. These alternatives are: Simpson's rule, which is the algorithm of choice (active at run level zero); the trapezoidal rule, which is less accurate than Simpson's Rule but is easier to calculate; and, the rectangle rule, which is less accurate than the trapezoidal rule but is computationally trivial.

\(^{15}\)The diagram given does not represent true SHADOW Script syntax. Details such as explicit internal variables have been removed to conserve space and reduce visual clutter for the purposes of this discussion.
Figure 6-8: A simple replacement example

The running time of each of these alternatives is a function of the size of the data sample taken from the curve. The sample dependency chain takes this into account as well, and provides two alternative ways of generating the data to be used during the numeric integration. The ideal technique generates a large data set based on high resolution sampling, while the alternative link produces a smaller data set with a much coarser resolution using less CPU time.

A quick inspection of the data flow graph abstraction might suggest indeterminant behavior or the potential for race conditions in situations where elements like the Process Area link seem to be taking its input from three other links simultaneously over a single data line. This is not the case, however. In each situation where multiple, alternative links exist, all alternatives hinge
off of the same activation flag, but use different, non-overlapping run level windows. For example, the trapezoidal rule link given in Figure 6-8 is only active at run levels three, four, and five. None of the other numeric integration links are eligible to be evaluated in this range. In the end, there can be only one link performing the integration activity at any given time.

Figure 6-9 illustrates the effect of various run levels on the actual flow of data through the graph presented in Figure 6-8. In each case the Process Area on the right side of the graph link was the start of the update exploration process. At run levels zero through two the subsystem performs its ideal processing thread, generating a high resolution data set and calculating the area under the curve using Simpson’s Rule. If performance starts to become a problem and the run level slips into the three to five range, the subsystem compensates by using the trapezoidal rule in place of the more computationally expensive Simpson’s rule, but uses the same high resolution data set as before. If system wide performance continues to degrade and the run level makes it into the six or seven range, the subsystem makes another compromise and begins using the less accurate rectangle rule. While the cost savings between using rectangles over trapezoids only amounts to one addition and one power of two division per data slice, at these run levels every clock cycle counts. Finally, if the run level gets dragged down into the eight or nine range the subsystem makes one final substitution and abandons high resolution sampling of the curve in favor a a rough sample and a small data set.
Figure 6-9: The effect of run levels on replacement decimation

As the above examples suggest, run level management under the SHADOW paradigm offers the user interface designer many opportunities for orderly, automated decimation of complex or time consuming tasks. The key to capitalizing on such opportunities, however, lies not in the infrastructure of the language, but in the designer's ability to recognize when alternatives should be provided and how individual links' run level eligibility windows should be tuned. Recognizing an opportunity for replacement is usually a function of one's familiarity with diverse algorithms and the priorities and goals of the overall system specification. Specifying the optimum set of thresholds for individual run level eligibility windows is somewhat akin to tuning an operating system and may require a combination of formal run time analysis, experimentation and empirical data.
VIRTUAL SUBSYSTEM MANAGEMENT

As was mentioned in the discussion of data abstraction, SHADOW subsystems are similar to object classes in object oriented programming languages. They provide a framework for specifying both data groupings and data manipulation methods, but do not actually constitute an executable system fragment until instantiated by the run time system. Additionally, there may be multiple instantiations of any given subsystem, each with its own private data area to differentiate it from its class siblings.

In most cases, subsystem instantiation is done during system initialization at the direction of the run time engine. This is also when the system wide dynamic data flow graph is built, the run time engine's task queues are constructed, and all augmented transition networks are initialized. However, the SHADOW system has been designed to support both static and dynamic allocation of subsystems through UIMS system calls.

Dynamic allocation of subsystems offers several benefits and conveniences to the user interface designer. First, if SHADOW subsystems are being used to model objects in a virtual world, use of virtual subsystems will allow those objects to be created and destroyed as needed by the application. Second, while the SHADOW was not intended for use as a generalized programming language, its support for dynamic subsystem allocation does allow it to chain subsystems together in dynamic data structures such as linked lists, heaps, and trees for performing large processing operations whose bounds are unknown at the time of program creation. Additionally, the use of a virtual subsystem simplifies the layout and increases the
clarity of data flow graph specifications by eliminating the need to explicitly interconnect multiple copies of particular subsystems.

The actual specification of a subsystem is the same whether its allocation method is static or dynamic, although a subsystem intended for dynamic allocation may include additional initialization code at the programmer’s discretion. The real specification difference arises when the subsystem is embedded in another module's data flow graph as a virtual link. Static links will either have a programmer specified activation flag or will be associated with the system flag ALWAYS. In contrast, a dynamic link will be specified with its activation flag set to the keyword VIRTUAL, indicating that until an instance of the subsystem is explicitly created, the link is not to be considered part of the dynamic data flow graph.

The connectivity for the virtual link is fully specified in the container subsystem’s data flow graph. Similarly, the programmer may specify other link attributes such as polling priority and run level windows but none of this information is taken into consideration until an actual subsystem has been created. Once one or more instantiations of the virtual subsystem are requested, the information specified for the virtual link will be used as a template for each and every instance’s attributes.

The actual allocation process is done via a UIMS system call from within the execution body of a link. While most attributes of the new subsystem’s interface will be inherited from the virtual link specification, the system call allows activation flags for each instantiation to be specified by the link which requests the allocation and may vary from one instance to the next. Additionally, a call used to instantiate a virtual subsystem also returns an identification tag
which may be used to deallocate the subsystem at a later time. Only dynamically allocated subsystems may be deallocated in this fashion.

An example of these principles applied to a simple problem is presented in Figure 6-10. In this specification fragment, a subsystem is responsible for managing a breakable vase in a virtual world. The goal of the subsystem is to place a model of a vase in a virtual world and, should a signal be received that the vase has been shattered, replace the model of the vase with dozens of shards modeling broken glass.

**Figure 6-10:** An example of virtual subsystem use
Note that the data flow graph of Figure 6-10 uses both virtual links and demand processing links to achieve its objectives. The two demand processing links are keyed to state transitions and are used to create and delete the virtual subsystems. Vase Manager link has been specified as a virtual to allow it to be deleted when the shatter signal comes through. Conversely, the Shard Manager link is specified as virtual to allow the creation of multiple shard subsystems in response to the same signal. All shard subsystems will need to know the position of the vase when it was broken as well as other environmental information (in this case coming from the window manager link). This information will be inherited from the virtual link definition.

The subsystem described in Figure 6-10 has the following behavior. Initially, the subsystem is created but no instances of either the vase or the shard subsystem exist. The system is activated, enters the initial state thus setting the START flag to TRUE. This, in turn, activates the Make Vase demand processing link. The Make Vase link instantiates a vase subsystem with an activation flag of ALWAYS, records the identification tag for the new subsystem in the variable VaseID, and then proceeds to deactivate its own flag and issue a DONE event. The augmented transition network proceeds to the Idle state.

The Idle state shown in Figure 6-10 does nothing except wait for an external signal which informs the subsystem that the vase has been shattered. When and if this signal, the OOPS event, is encountered, the subsystem enters the Broken state, setting the SHATTER activation flag to TRUE.

When its activation flag is triggered, the MakeShards demand processing link is evaluated. This link uses the VaseID value to delete the vase subsystem and creates dozens of instances of
the shard subsystem in the vase's place. The initial position for each shard may either be a
random offset from the vase's position or may be coordinated by the underlying application
depending on the shard subsystem's implementation.

**GENERIC CONSTRAINT SUBSYSTEMS**

Generic constraint subsystems were designed to provide the SHADOW system with a means
by which the data flow specifications for individual subsystems could be simplified: and
generalized behavior could be captured, specified and applied to a variety of user interface
elements. Each generic constraint has a set of internal scoping rules which determine when
they should be applied by the run time engine as well as a task list which enumerates those
subsystems that are valid candidates for processing with said scope. Internally, a generic
constraint subsystem looks very much like any other SHADOW module, with the exception
that its data flow graph always begins and ends with a set of property variables, which
constitute its sole link to the balance of the UIMS\textsuperscript{16}.

The initial scope of a generic constraint is a function of its location in the containment hierarchy
tree. Generic constraints do not form nodes in the tree by and of themselves, they must be
associated with particular subsystems or be specified in the interface application's system
definition (in this latter case the generic constraint becomes global to the application). When a
generic constraint is included by a module, it has the potential to apply to every other
subsystem directly or indirectly contained by that module. Likewise, if virtual subsystems are

\textsuperscript{16}The actual specification of generic constraint is discussed in detail in Section IV: The SHADOW Talk
Model.
created within such a hierarchy tree branch, they, too, will be subject to the generic constraint's specification\textsuperscript{17}. Figure 6-11 illustrates how these scopes propagate through a containment hierarchy tree.

\textbf{Figure 6-11:} Generic constraint scoping within the containment hierarchy

When creating a generic constraint, the designer should be aware of several factors with respect to how the constraint will be applied by the run time system. First, it is possible for multiple generic constraints to be acting on any given subsystem at any given time. All such interactions will be conducted via alterations the subsystem's property values. Due to the structure of the containment hierarchy and the potential for code reuse, the individual constraints may not be aware of one another's existence or behavior. In light of this, a designer should create generic constraints which either isolate their effects on objects by specifying associate properties, or apply their effects on core properties as incremental deltas, updating old values rather than summarily replacing them.

\textsuperscript{17}This is not to say that all subsystems are affected by the constraint simply by virtue of their position in the tree. Each constraint is further scoped by use of a core property mask. A subsystem must possess each and every property specified in the mask to be affected by the generic constraint.
Second, the order in which individual generic constraints are applied to subsystems is neither fixed nor predictable. In cases where multiple constraints are to act upon the same subsystem, the designer should avoid making assumptions regarding which set of rules will be applied first or what manner of transformations have already been performed on a particular property value.

Finally, the frequency with which generic constraints are applied is determined by the run time system and is beyond the designer’s ability to dictate or predict. The designer should avoid the assumption that constraints will be applied uniformly and unilaterally. Where the real elapse time between applications of a generic constraint is an issue, the constraint should define an associate property to record the time of the most recent update and internal processing of the constraint should be structured around the time delta between the current time and this value.

As a slightly contrived example of the application of these principles, consider a system which approximates the vertical motion of a rocket using a passive subsystem and two generic constraints. The subsystem is simply an abstraction for the rocket itself which maintains property values for its altitude, velocity, acceleration, weight, fuel supply, engine status, and thrust to weight ratio. One generic constraint keys off of all of the properties and, if the engines are on, updates the position and motion vector of the rocket as a function of the engines’ thrust as well as decreasing the fuel supply and overall weight as a function of the burn’s duration. The other constraint simulates the pull of gravity on the rocket and determines how far the rocket would have fallen in free fall given the current motion vector over a brief period. Each constraint appends its own time stamp information to the subsystem so it knows how large a time step to apply on each application. While the potential for large time steps,
variable acceleration and variable weight of the rocket suggest that such a problem breakdown would not be accurately predicting the position of the rocket at each and every instant in time; the overall approximation of the motion creates an illusion of realism with the virtual world, and the generic constraints themselves could equally well be applied to dozens of rockets with varying properties as well as random freely falling bodies with little or no modification.

**SPECIFYING LINK INTERNALS**

In order to allow a high degree of flexibility without introducing yet another syntax into the domain of procedural languages, the SHADOW system uses a subset of the C++ programming language for specifying the internal execution bodies of link elements. Within a link body, a programmer may use all control flow and assignment operators normally available when programming in C++. Calls to external library functions and references to externally defined types and object classes may be made, provided that appropriate header inclusion information is provided to the containing subsystem. However, the execution body of a link should be thought of as a code fragment within a larger function and, as such, the program may not specify additional type definitions or subroutines within this code space. Additionally, a link execution body should never contain a return statement; the presence of such a statement is always an error and may result in data corruption.

When defining a link, the designer specifies a set of input variables and output variables. This constitutes the limits of the programmer's ability to control the 'parameter list' of the link's function. All data is passed to the link by value and may be changed locally without effecting
the data flow graph. All variables are identified by name and additional, internal variables may be specified. Care should be taken to ensure that all defined output variables are assigned a return value within the code, failure to do so may lead to unpredictable results.

Within the confines of the run time engine, the body of a link represents an atomic unit of processing. While the UIMS has been structured to support deadline based calculations and automated decimation, it has no ability to preempt the evaluation of a link's execution body once an update has begun. Additionally, unrestricted use of C++ code fragments, and the ability to make calls outside the domain of the UIMS present several opportunities for bad programming practices which can reduce or eliminate the run time engine's ability to meet its performance goals. As a general rule of thumb, large iteration structures such as nested loops should be avoided within link definitions. Additionally, sequential calls to computationally intensive, time consuming functions should be eliminated. The functional goals of such routines may be achieved by either distributing the processing load over several link definitions (allowing the run time engine the opportunity to decide how much processing may be done in the given iteration time step) or by farming tasks off to separate operating system level processes which can be executed concurrently with the UIMS itself.

To simplify the process of integrating C++ language procedures with the SHADOW specification paradigm the UIMS provide a library of system calls design to access many of the run time engine's functions from within the execution body of a link. This library is included automatically for all links (as is the stdio.h specification). This library includes: functions to request the creation and destruction of virtual subsystems; functions which allow links to RAISE, BROADCAST and directly PASS event tokens from one subsystem to another; and,
functions for sending update request signals across dependency threads within the data flow graph. A detailed description of the exact calls, their parameters and their return values may be found in Appendix B.

**DEVICE MANAGEMENT**

The field of non-WIMP user interface development is a domain rich with novel and customized data input and output devices. While such devices are both beneficial and essential tools for exploring the complexities of human-computer interaction, the dynamic nature of the devices' designs and the lack of standards regarding their implementations introduce an extra level of complexity into the development and maintenance of any software system built around such devices.

In accordance with the design philosophy adopted by development languages like C and C++, the SHADOW System makes no assumptions about the number or types of I/O channels that will be introduced in the course of interface specification. Instead, it supports a mechanism by which new device handlers may be built, integrated and reused as needed in future developments. This approach has both benefits and drawbacks. The primary drawback of the system is the need to create SHADOW compatible device drivers or device driver wrappers for each and every new device introduced. The benefits include: the ability to extend and evolve the I/O facilities of the UIMS in a consistent manner indefinitely; the ability to create abstract interfaces which isolate device control from the balance of the interface specification; and, the ability to amortize the cost of device driver development over multiple efforts by means of code reuse.
The SHADOW System supports two specialty classes of links intended to facilitate the creation of interfaces to external devices. These are the INPUT and OUTPUT class links. Like normal PROCESSING links, these elements have individual activation flags and run level eligibility windows. They may be turned on and off by accompanying augmented transition graphs or by the run time engine itself in the course of decimation processing, and may be wired into a subsystem's data flow graph in the same manner as any other link.

Depending on the nature of the device and interaction desired, entity-specific INPUT and OUTPUT links may either be centralized into device driver subsystems or utilized as simple utility calls dispersed throughout many subsystems. For example, a head-mounted display equipped with an eye-tracker may be integrated into a SHADOW Talk specification by creating a subsystem for reading all pertinent data from the device and packaging that data for export into the application's dynamic data flow graph. Internally, this subsystem includes an INPUT link which reads the spacial tracking information from an external daemon executing as a parallel process, as well as a second link which accesses an area of shared memory to obtain eye-tracking information provided by yet another external process. Externally, this subsystem functions as a single unit with several output terminals which may be connected to other links and subsystems to satisfy the processing needs of the interface.

Conversely, there are some situations where centralizing the control of an external device within the UIMS may represent an inconvenience. For example, an application may make use of a standard graphics library (such as the Xlib drawing primitives), and forcing all drawing
requests through a single internal subsystem may be convoluted and counter intuitive. In this case, the external library itself may be considered the central processing point, and each subsystem which desires to draw simply maintains its own OUTPUT link which includes calls to the external library within its execution body.

Centralizing device control within a subsystem also centralizes the application’s dependency on the device in question into a more manageable area. If the device should be modified or replaced, such that its external application programming interface should change, all volatile links will be clustered into a single module. So long as the effected module’s interface with the dynamic data flow graph can be preserved, the external device modification will not require system-wide coding modifications.

In contrast, if a decentralized approach is taken, and many subsystems contain links which make calls into external libraries, any modification of those libraries has the potential to trigger a maintenance ripple effect. All subsystems built upon the effected library could potentially require modification. Each link with the targeted subsystem would need to be verified to ensure that all external interactions are conducted in accordance with the current interface specification.

If a centralized approach to device management is selected, a dedicated device driver subsystem may be constructed in a manner identical to any other SHADOW module. For example, a subsystem for controlling three-space mouse might contain an INPUT link which periodically polls a direct memory address area on a given machine. This operation returns: values reporting the spacial position of the mouse with respect to some point in space; the orientation of the mouse with respect to the force of gravity; and, the status of the mouse button as either
depressed or free. The subsystem may also include other links to process the raw data into more convenient formats such as changing the frame of reference for the position data and converting the button status data to a set of button transition events to be raised to its parent module in the containment hierarchy.

The heart of such a subsystem relies on the INPUT link. Whether used as part of a dedicated device driver subsystem or distributed throughout an interface specification, the INPUT link occupies a key role in the update process of the interface’s dynamic data flow graphs. In the course of exploring a link’s dependencies, if it is found that the input data to a PROCESSING or OUTPUT link has not changed since the link’s last update cycle, the link is not evaluated. This is not the case with INPUT links (which have the potential not to specify any plugboard data dependencies at all). Any time an active and run level eligible INPUT class link is encountered in the course of an evaluation exploration, it is guaranteed to be updated. Additionally, INPUT class links may be polled directly by means of an update priority code to ensure that data from external devices will continue to be monitored even if no other active subsystem in the application has an immediate need for the output data of the link\(^\text{18}\).

Output device driver subsystems are very similar to input channels in that they consist of a SHADOW Talk based interface, and contain links which either perform internal processing or talk directly with external entities. In this case, the key elements of these subsystems are the OUTPUT class links. These links are the elements responsible for interfacing the SHADOW UIMS with external graphics libraries, rendering engines, sound generators, tactile feedback systems, etc.

\(^{18}\)This technique can be quite beneficial when input data processing involves tokenizing a continuous data stream into discrete event signals.
As individual subsystems are instantiated and merged into the interface application's dynamic data flow graph, the location of all OUTPUT class links is recorded and used as the basis for the run time engine's output generation cycle task list. Once per system iteration step the system will query each OUTPUT link on file and determine if current circumstances warrant updating the link and its dependencies. If the OUTPUT link is designated as being active at the current run level, then it is used as the starting point for a data flow graph evaluation exploration. If, as a result of this exploration, the data upon which the link depends has changed since its last activation, the link will be updated.

**CLOSING COMMENTS**

The topics and techniques presented in this section illustrate how the SHADOW user interface description language and its run time engine may be used to form the infrastructure around which non-WIMP style interfaces may be built. In the sections that follow, this claim will be empirically tested. Complete applications designed to test the features of the UIMS and created within the SHADOW specification paradigm will be presented.

Due to the current lack of a robust graphical editor, the SHADOW Talk diagrams presented in Sections VII and VIII were generated and manually converted to SHADOW Script source files. This conversion process was purely clerical and mechanical in nature and eventually will be completely automated by the visual editor (currently in development). The resulting source code was fed to the SHADOW System's compiler which automatically generated the executable files depicted and discussed in the text.
The lessons learned from this development activity will be used as an indicator of the SHADO
system's ability to satisfy the criteria for a generalized specification paradigm suitable for application to the non-WIMP style interface domain as defined in Section III.
SECTION VII: EVALUATING THE SHADOW PARADIGM

OVERVIEW

The SHADOW System was designed to address a variety of issues inherent in the specification and implementation of non-WIMP user interfaces. Its level of success with respect to these goals can be measured indirectly by attempting to create applications which rely upon the purported features of the UIMS. The lessons learned from the development exercises, combined with a concrete measure of the success or failure of the applications to perform as intended, may be interpreted as an indicator of the success or failure of the UIMS itself.

To date, fifteen complete applications and a variety of code fragments have been developed to test the correctness and robustness of the SHADOW System in general, and the specification language in particular. From this pool of applications, several sample development efforts will be presented in detail in this section. The first will be the obligatory "Hello, World" program as written within the SHADOW paradigm. This will be followed by four examples which highlight individual features of the UIMS. The goal of these applications is to demonstrate the SHADOW System's ability to substantiate its claims of support for modular development, parallel/concurrent processing, discrete and continuous processing, event abstraction, integration of event controlled, constraint declarative programming, etc. The size and complexity of these sample endeavors has been deliberately scoped to create a forum which is both large enough to be meaningful and small enough to effect unit tests on individual features.
of the SHADOW paradigm and the UIMS. Once the base claims of the system have been empirically verified, a large scale integration test will be presented in the form of the creation of a complete virtual world application. A detailed discussion of the virtual world development effort, however, will be deferred until Section VIII.

A TRIVIAL APPLICATION

As Kernighan and Ritchie pointed out in The C Programming Language [83], the basic hurdle that must be crossed for any new language is the development of the classic "Hello, World" application. While the complexity of the program itself is trivial, the ability to design, implement, compile and run such a stand-alone application provides evidence that the development tools and the run time engine function as designed. This is the first step toward testing whether or not the said tools function as intended.

The annotated SHADOW Talk source for the Hello, World application is shown in Figure 7-1. In essence, this system consists of two DEMAND class links and a simple state diagram which controls their activations. The first link is an OUTPUT link called Start, keyed to be evaluated exactly once each time the flag START is set to TRUE. The body of this link is a simple C printf() statement which displays the actual greeting to the user via the stdout channel. The second link is a PROCESSING link which is evaluated exactly once each time the flag END is set to TRUE. The body of this link consists of an invocation of the C library call exit().
System: Hello
Input Events: (none)
Output Events: (Not Applicable to Systems)

This system implements the SHADOW-Talk equivalent of the classic HelloWorld.c example. It consists of a DEMAND OUTPUT link called Start which is evaluated exactly once, each time the flag START is set to TRUE. The body of this link is a simple C printf() statement. The second link is a DEMAND PROCESSING link which is evaluated exactly once each time the flag END is set to TRUE. The body of this link consists to the C library call exit().

The setting and clearing of the activation flags for the links is handled by a state diagram. The diagram consists of two states (StartState and EndState) with the initial state of the system indicated by a double ringed circle. In StartState, the flag START is set to TRUE, END is set to FALSE and the system waits for the event DONE to signal a transition to EndState. (The event DONE is generated automatically by the run time engine whenever a DEMAND link is evaluated.) Once such a token is encountered the system enters EndState, the flag START is set to FALSE, END is set to TRUE (allowing the link End to be evaluated) and the system ignores any future events (such as End link's DONE signal). In this case the evaluation of the End link causes the program to self terminate.

![State Diagram]

**Figure 7-1:** The SHADOW Talk answer to "Hello, World!"

The setting and clearing of the activation flags for the links is handled by the state diagram. The diagram consists of two states with the initial state of the system indicated by a double ringed circle. In StartState, the flag START is set to TRUE, END is set to FALSE and the system waits for the event DONE, generated automatically by the run time engine whenever a
DEMAND link is evaluated, to signal a transition to EndState. Once such a token is encountered, the system enters EndState, the flag START is set to FALSE, END is set to TRUE.

In practice, this specification allows each link to be evaluated exactly once. When StartState is initially entered, the flag START is set to TRUE, triggering the demand evaluation of the link Start. When the link Start has completed its task of printing the greeting it broadcasts a DONE event, forcing the system to make a transition to EndState. Upon entering EndState the flag END is set to true, triggering the demand evaluation of the link End, which terminates the process with a system call.

On a first inspection of the "Hello, World" application, the visual language of SHADOW Talk and its elaborate run time engine may seem to be a very cumbersome means of specifying the equivalent of a two-line C program. However, it is important to remember that this small, discrete, serial, asynchronous, half-duplex application is about as far removed from the intended purposes of the SHADOW System as one can get while still generating a complete application. In essence, applying the SHADOW System to this class of problem incurs all of the overhead\textsuperscript{19} of the UIMS without realizing any of the potential benefits the system has to offer.

Even in the context of a trivial program, however, certain characteristics of the SHADOW paradigm begin to manifest. Conceptually speaking, the application consists of two discrete tasks: one to print a message; and, one to terminate the program. Additionally, these tasks are disjoint with respect to both data utilization and temporal execution space. These conceptual

\textsuperscript{19}In addition to requiring the creation of a UIDL specification which provides no added value, such an application wastes system resources. The size of the final executable for this application was 57 kilobytes, nearly all of which was the binary for the SHADOW RTE.
properties are directly reflected in the syntax and semantics of the SHADOW Talk specification. The two tasks appear as separate links. The lack of data interdependence is shown by a lack of connective flow lines between the links of the data flow graph. Inspection of the augmented transition network reveals that the flags which control the activation of the two tasks are mutually exclusive, ensuring that the second task may only begin after the first has run to completion.

The success of the "Hello, World" application demonstrated the ability of the SHADOW UIMS to correctly compile and execute a SHADOW Talk specification residing entirely within a single file and performing exclusively discrete operations. While emotionally reassuring, the exercise did little to explore or exploit the bounds of the SHADOW System's purported capabilities such as integrating continuous and discrete data relationships, support for concurrent/parallel processing activities, support for programming in the large, and, in general, differentiating itself as a viable tool for the development of non-WIMP user interfaces. Each of these claims will be explored in greater detail in the examples that follow.

**DEVELOPMENT OF A WIDGET SET**

Unlike most user interface management systems, the SHADOW System does not have a fixed widget set of commonly used interaction objects associated with it. This is largely due to the fact that there is not yet a strong consensus on how the concept of a widget ports into the domain of virtual reality and other non-WIMP arenas. While common GUI push buttons, scrollbars, and typing fields may all be easily created within a virtual world, the appropriateness of such artifacts within such a medium is open to debate. The SHADOW System sidesteps this debate by providing mechanisms by which reusable, loosely coupled
Code fragments which perform commonly desired user interaction tasks\textsuperscript{20} may be specified and used as the building blocks for larger functions and applications.

Beyond the immediate scope of user interface development, the basic concept of structuring more complex from simpler units (or modules) can have far-reaching ramifications with respect to the applicability of a language to large scale development efforts and to the long term viability of systems developed under said language [8][92][93]. Conversely, failure to address issues of scale and modularity, especially within the domain of visual languages, may lead to systems which, while contributing valuable concepts, are either impractical to apply, or cumbersome to maintain or internalize beyond the complexity domain of trivial applications.

Several applications were developed to test the claim of support for modular development and code reuse. One of these efforts focused on the creation of a traditional WIMP-style widget set. The task was undertaken not so much for the utility of the widgets themselves, but rather as a proof of concept exercise for the tools and techniques needed to create a generic library of interface building blocks which could support modular development. The innate nature of many of the widgets created also afforded the opportunity to explore other claims of the SHADOW UIMS, in particular: the ability to interface to external input and output channels (X windows); the ability to create, propagate, consume and abstract discrete event information; the ability to bind and unbind continuous data relationships; and, the ability to perform minimal (lazy) evaluations of a dynamic data flow graph.

\textsuperscript{20}One could argue that this is the essential definition of a widget. Development work done on the Rookery application (discussed in Section VIII) suggests that within the confines of a virtual world, the optimum 'widget set' for a developer is not a palette that mimics the cut and paste objects found in current WIMP interfaces (such as Button and Slider) but rather a toolkit of behaviors and abilities (such as Graspable, Throwable, and Bendable) that may be associated with elements of the virtual world as needed.
A detailed exploration of the development process and finished product of each and every widget in the final library would be both space prohibitive for this work and hideously dull. Instead, a small subset of key examples from this effort will be presented. These examples have been selected for their ability to highlight many of the language features mentioned above.

Perhaps one of the simplest interaction objects to surface from the WIMP domain is the push button. Conceptually, this is nothing more than an area of the screen which presents some form of graphical cue to the user as the the button’s existence and function, and an event handler which triggers some form of customized processing on the programmer’s behalf when it detects that it has been activated. A SHADOW Talk specification for this class of widget is given in Figure 7-2.

Analysis of the plugboard reveals that the button consists of three potential processing tasks: two possible variations for the button to be drawn; and, a demand processing link which functions as the trigger for a callback operation to function somewhere in the underlying application. The plugboard also records the position, geometry, and generic appearance of the button (used by the drawing links) as well as a dependency thread (Refresh) to ensure that any instance of the button is redrawn when whenever the button’s containing supersystem declares a need (such as an X Window Expose event). Additionally, the plugboard provides inputs for the location of a pointing device (Mx and My). Although these values are not used by the data flow graph itself, their declaration is required so that they may be referenced by the condition statements of the associated augmented transition network.
Subsystem: PushButton
Input Events: MouseUp, MouseDown
Output Events: ButtonPressed

Figure 7-2: A specification for a push button widget
The augment transition network for the push button shows that the widget has three states: IdleState, meaning that the button is waiting to be pressed; ArmedState, indicating that the button has been pressed and that the widget is waiting for the button to be released before activating the callback operation; and ActiveState, in which the user action sequence of press and release tokens have both occurred within the geometric bounds of the widget and the callback function is marked for activation. By default, the initial state for this network is IdleState.

The behavior of this widget is strongly event driven and, as such, most of its computational complexity can be found in its event handling and transition network. The subsystem listens for two instances of external events, MouseUp and MouseDown\textsuperscript{21}. This discrete information is combined with continuous local data to form the conditional state transitions of the ATN. A mouse click alone cannot trigger the button to fire unless the action occurred within the current bounds of the button itself (this check is performed by the external function $\text{Inside}(\ldots)$).

In the event that the button does indeed fire, the callback link will be evaluated exactly once. The body of this link performs two functions. First, if the property Callback has been set to any non-NULL value, it is assumed to be the address of a callback function in the underlying application and the function is called. Second, a new event, ButtonPressed, is generated and propagated up to the containing supersystem. This scheme allows the button to function as both a dialog control element for the underlying application as well as an event abstraction element for complex processing within the SHADOW domain.

\textsuperscript{21}In practice, these events are generated by an X Window interface subsystem but their origin is unimportant in the context of this specification.
The second element of note in the widget set is a horizontal, proportional slider. This consists of a background trough of a given width and height in a certain position on the screen and a slider handle whose size is a function of the ratio between the active part of some underlying data set and the total size of that set. For example, if the slider were associated with a list of ten elements but only three of the elements were visible on the screen, then the slider handle would be 30% of the width of the total area. The position of the handle within the trough is a function of the position of the first active data element within the associated range. The make up of such an element is shown in Figure 7-3.

\[ \text{(XPos, YPos)} \]
\[ \text{Size} \]
\[ \text{Width} \]
\[ \text{Height} \]
\[ \text{HandlePos} \]

**Figure 7-3:** The anatomy of a proportional slider widget

The operation of the slider widget is relatively simple and lends itself to specification and implementation within a constraint based system. When the user grabs the handle of the slider, the position of the slider within the trough becomes a direct function of the pointer's horizontal location. Likewise, the index value of the active window in the underlying data set is constrained to the value of the slider position\(^2\). Determination of the handle size is a simple

---

\(^2\text{This constraint is actually bi-directional. If the index of the active data window within the associated data set should change due to external circumstances, the slider handle must reposition itself accordingly within the trough.}\)
transformation which takes the ratio of the active data window's size versus the data set size and scales the result by the width of the trough.

Converting this natural language description of the widget into an actual SHADOW Talk specification results in the subsystem given in Figure 7-4 and is strongly reminiscent of the simple slider specification predicted in Figure 4-8. The operation of the widget consists of four processing tasks, two of which are always active and two which hinge off of user actions. The two perpetually active links of the plugboard are: HandlePos, which is responsible for calculating both the position of the handle and its width as functions of the widget's geometry and the associated data set (DataPtr); and DrawSlider, which performs basic rendering functions whenever the last rendering of the slider becomes obsolete. These links remain active whenever the subsystem itself is active and respond to changes in widget size or data set size automatically.

The links InitDrag and ValueDrag become eligible for recalculation only in response to the user grabbing the slider handle (in practice, this is done by pressing the left mouse button over the handle and dragging to a new location). InitDrag is evaluated exactly once and is used to define a fixed offset between the initial position of the mouse and the handle's home position when the operation began. This offset is used by ValueDrag as it calculates how changes in the mouse's horizontal position should relate to the index of the active data window within the associated set and ultimately to the position of the handle within the trough. The ValueDrag link remains active for as long as the left mouse button is held down, even if the mouse pointer subsequently leaves the screen area of the widget.
Subsystem: HPSlider
Input Events: MouseUp, MouseDown
Output Events: (none)

Figure 7-4: A specification for a horizontal slider with a proportionally sized handle
While the iconology and connectivity complexity of Figure 7-4 may seem a bit overwhelming to the uninitiated at first, both the specification and implementation efforts amounted to trivial activities (the most challenging task being the reduction of flow line crossings in the diagram layout itself, which is more a function of pretty printing than of processing). In a sample test application the slider was associated with a sliding window onto an array of numbers. Initially the array size was equal to the window size. This resulted in the handle being positioned at the zero mark of the trough and filling the entire width of the widget's space. As new elements were appended to the data set externally the handle's width shrunk proportionately while remaining at the same starting position. When items were externally removed from the end of the array and prepended to the beginning (effectively moving the active window through the data set one element at a time) the handle position shifted accordingly and automatically. In response to user intervention, both dragging the handle and resizing the slider itself, the widget responded as desired and expected both resizing and repositioning the handle as well as modifying the index of the active window within the associated data set.

The third widget of note in the sample widget set illustrates the ability of the SHADOW System to support modular development and helps solidify the belief that push button and slider presented above are in fact widgets and not simply contrived examples with little or no potential for reuse. The scrollbar shown in Figure 7-5 is a composite widget made of two push button widgets (with differing icons) and a horizontal, proportional slider widget23.

23This same technique of composite widget construction is commonly used in traditional WIMP-style toolkits.
Figure 7-5: The composition of buttons and a slider to form a scrollbar widget

Like the slider itself, the scrollbar is intended to be associated with some underlying data set of which only a small window of which is 'visible' at any given time. Again, the position of the handle is both an indicator and a control as to where the window falls within the data set and the size of the handle is an indicator as to the size of the window with respect to the data set as a whole. Unlike the slider, which deals almost exclusively in continuous relationships to perform its functions, the scrollbar adds two discrete event elements in the form of push buttons to allow the user to force the active window to flip backwards or forwards a page at a time (repositioning the slider handle in the process).
Subsystem: HPScreenbar
Input Events: MouseUp, MouseDown
Output Events: (none)

![Diagram of a horizontal scrollbar with a proportional handle]

**Figure 7-6:** A specification for a horizontal scrollbar with a proportionally sized handle
The SHADOW Talk specification for such an element is given in Figure 7-6. The widget consists of three contained subsystems (two instances of PushButton and one of HPSlider) which perform most of the functionality of the widget. In addition to these elements, there are two demand processing links which are fired in response to ButtonPressed events from the two arrow buttons. These are the elements that perform the actual page flip for the active window index with the data set. The slider subsystem is not aware of who has changed the window information; only that a delta has occurred and updates its own renderings accordingly the next time its output is needed. The three remaining links of the plugboard are simple layout constraints for managing screen real estate in the face of resizable windows and contribute little to the functionality of the widget other than appearance.

As with the slider widget, the scrollbar was specified and implemented using SHADOW and tested against a trivial application. In addition to functioning as expected, the widget specification involved very little effort compared to the development of the proportional slider. This can be attributed to the fact that most of the conceptual and syntactic details of the specification had already been addressed in the development of the slider and could be reused directly by embedding a slider subsystem within the scrollbar specification.

The ease of development and successful transformation of the scrollbar widget serves to support the claim that modular development within the SHADOW paradigm is technically feasible and suggests that there are benefits to be realized when modular development practices are followed. The fourth and final example from the widget set will take this argument one step further and demonstrate that the benefits realized in the development of the scrollbar are not an isolated example and that scalability through modularity, data hiding, and hierarchical abstraction are inherent properties of the SHADOW paradigm.
The final widget to be presented in this section is a specification for a scrolled area. This class of widget forms the basis for many windows found in today's graphical user interfaces. Conceptually, it consists of a screen area which serves as a viewport onto a larger virtual space, such as a text document or drawing table. The mapping of this window onto the underlying area is controlled by a pair of scrollbars, one horizontal and one vertical.

Just as the scrollbar was composed of two simpler subsystems and a few layout and integration links, the scrolled region subsystem can be built from lower level primitives. One such subsystem is the horizontal scrollbar itself. The other is a vertical, proportional scrollbar whose design and implementation is virtually identical to that of the horizontal scrollbar with a simple transposition of the X and Y values in the governing logic. An example of a scrolled area widget based on these two subsystems is given in Figure 7-7.

As the level of abstractions grow, the focus of the specifications migrates away from the mundane details of how to conduct low level processing or graphical presentation and converges on issues of data packaging and routing within the data flow graph itself. This can be seen both in the scrollbar example and in the scrolled area specification. In the latter case, the largest amount of processing done by any of the three local links occurs in BundleVP, whose sole purpose is to consolidate a variety of low level data items into higher level abstractions for ease of reference within the contained subsystems.
Subsystem: ScrolledArea
Input Events: MouseUp, MouseDown
Output Events: (none)

Figure 7-7: A specification for scrolled screen region
A wide assortment of widgets was created in the same manner as those presented here, including labels, push buttons, toggle buttons, value selection sliders, range sliders, proportional allocation sliders, pull down menus, pop-up dialog boxes, radio boxes, scrolled pick lists, and single and multi-line text entry fields. Throughout all, the SHADOW System's support for modular development and event abstraction was found to be a great benefit both from the standpoint of reducing development time and effort as well as limiting the visual complexity of the individual specifications. Additionally, the design effort associated with several widget classes, such as the proportional slider and generic container widgets, was dramatically reduced by utilizing the strengths of the plugboard data flow constraint specifications. In the case of highly discretized operations, such as text field editing, the event handling facilities of the SHADOW System were found to be more than adequate for the tasks at hand.

DEVELOPMENT OF A TEXT EDITOR

The successful development of a widget set within the SHADOW paradigm illustrated the ability of the system to support modular programming and code reuse at the component level. Additionally, the lessons learned from the development effort itself suggested that there were distinct benefits to be realized from appropriate use of SHADOW Talk's event-controlled, data constraint specification capabilities. In addition, the ease of creating composite widgets and the resulting levels of complexity in the specifications suggested that the SHADOW paradigm is indeed scalable to large programming projects without collapsing under the weight of its own visual notation.
To test this hypothesis, a text editor application, Notepad, was developed using the widgets previously discussed. The finished product of this effort is shown in Figure 7-8.

![Generic makefile for the SHADOW Application](image)

**Figure 7-8:** A simple text editor built from SHADOW WIMP widgets

A text editor was chosen as a preliminary stress test for both scalability and robustness of the SHADOW paradigm, not because the SHADOW System is well suited for such developments, but because a text editor, with its single, sequential stream of discrete tokens and ping-pong
style dialogs [63], is nearly a complete opposite of the type of application the SHADOW System was optimized to support. The goal of the effort was to demonstrate that the underlying foundations of the SHADOW Talk language hold even when applied to tasks which lie outside of its intended application domain. Additionally, the creation of a text editor application provided an opportunity to perform tests on the run time engine itself to ensure that the overhead incurred by the system's data flow graph maintenance did not severely compromise the application's ability to read and service bursts of discrete event tokens (keystrokes).

The data flow graph for the final Notepad application is given in Figure 7-9. While somewhat sprawling in area, the graph itself is not very complex and clearly illustrates the fact the the system is composed of five major subsystems and a handful of local links. The five subsystems divide the functionality of the editor into several parallel domains. The Display link encapsulates a subsystem designed for communicating with X Windows, opening a display channel and converting X event queue entries into SHADOW events and data values. The Dialogs link connects the Notepad system with its only custom subsystem, a dialog manager designed to control the position, appearance and event masking properties of the editor's pop-up dialog boxes. The Window link encapsulates a scrolled text editor widget which forms the backbone of the application's processing power. The Menu link introduces a generic menu bar with a pull down file menu taken from the widget set and the TitleBox link simply places a label at the top of the display to identify the application.
Figure 7-9: The plugboard specification for the Notepad application.
Of the local links, only the dynamic processing link, LayoutDisplay, has any internal processing of significance. This link is responsible for initially defining the positions of the various elements within the overall window area and for revising those definitions in the event of an external window resize event. The other three local links are all demand processing links used simply for event abstraction to send specific signals to the subsystems and to force transitions in the associated transition network.
The state diagram associated with the plugboard of Figure 7-9 is given in Figure 7-10. The six states of this network describe, at the highest level, the macro behavior of the text editor. The initial state, StartState, is used to activate the Display and LayoutDisplay link while keeping the others dormant until Display has succeeded in opening up a communications channel to the local X server. When the display is ready, the XWindow subsystem sends a ScreenReady event to trigger a state transition to RunState. In this state, the other contained subsystems become active, enforcing constraints and servicing events each according to its own local definitions. The use of the state Popup is somewhat unique in the diagram in that its purpose is not to enable a function but rather to disable the menu and editor so that pop-up dialogs may take exclusive control of the event queues and screen real estate. The remaining states are used to fire demand processing links in response to events raised by the menu bar subsystem.

The augmented transition network of Figure 7-10 is both deceptively simple for the true complexity of the Notepad application and complete with respect to the level of detail the designer needs to be cognizant of at this level of the specification. Each of the subsystems contained in the Notepad system definition maintains its own augmented transition network, as do all of the other subsystems contained in turn. This technique of nesting and hiding of state information has proven to be instrumental in managing the cognitive load of large and complex applications when applying the SHADOW System.

The actual development of the Notepad application took less than an hour\(^{24}\) to complete and served to support the claim that a graphical language can be applied to full scale applications without becoming overly cluttered or complex. In the course of that development, however, an

---

\(^{24}\)This figure does not reflect the two weeks' worth of effort needed to create the basic widget library from which the application was derived.
Idiosyncrasy surfaced with respect to the handling of events and parallel evaluation of links when integrating various elements of the widget set.

Figure 7-11: Screenshot of the Notepad application showing menu and dialog widgets

Figure 7-11 shows the Notepad application with one of its pop-up dialog boxes open. In the first attempt at creating the application it was discovered that if one naively creates buttons and dialog boxes as one would with Motif or OpenLook, the parallel evaluation engine and event propagation policy allows events such as a mouse click to be serviced by multiple widgets concurrently. In the case of the dialog shown in Figure 7-11, clicking on the Okay button would also trigger the cursor of the editor window to be repositioned to the location of the click. While the fix for this bug was simply a matter of using the Popup state of Figure 7-10 to turn off the other widgets' container links, the exercise did demonstrate that in providing the flexibility to create, abstract and consume events at will, the SHADOW System has also introduced the potential to require developers to explicitly address certain tasks that had once been taken for granted.
With the exception of the visual layering problem discussed above, the creation of the Notepad application was both simple and straightforward requiring the creation of only two SHADOW modules. The first being the Notepad system itself and the second being a dialog manager subsystem. Like the Notepad system, this module used a network of generic widget subsystems to perform low level operations and demand processing links to hand off tasks from one subsystem to the next. Again, the augmented transition network was used as the primary control mechanism for managing discrete sequential processing tasks. For completion, the specification of this subsystem is given in Figure 7-12.

25To put this issue in context, recall that the SHADOW System was never intended as a replacement for traditional WIMP GUI Builder systems and that generalized event handling and parallel processing are more valued abilities than sequential event processing within a non-WIMP application domain.
Subsystem: Dialogs

Input Events: OpenRequested, SaveAsRequested

Output Events: DialogClosed

Figure 7-12: The popup dialog box manager for the Notepad application
PROGRAMMING CONSTRAINT DRIVEN BEHAVIOR

The SHADOW Talk language is not simply a graphical wrapper for visually programming C++ routines. The SHADOW System offers a unique programming paradigm with its own costs and benefits which, at times, may be quite removed from the mind set of common procedural programming. The fourth application to be presented in this section was selected both to exercise some of the more esoteric features of the SHADOW System and to illustrate the alternate design approaches possible under this paradigm.

Figure 7-13 presents a screen capture taken from a simple video game developed using the SHADOW System. Conceptually, the game consists of: a paddle, which the user can move back and forth along the bottom of the screen; a ball, which is always in motion and follows a linear path until blocked by some other object, at which point it bounces off on a new trajectory; sidewalls and a ceiling, which prevent the ball from flying off the screen on three of the four edges; and, an array of bricks, each of which may block the ball but will be destroyed in the process. The purpose of the game is to remove all the bricks (by bouncing the ball off of them individually) while preventing the ball from escaping off the bottom of the screen with the paddle.

The game itself is not a terribly original construct. Games like this have been around for more than twenty years in one form or another. This vintage arcade game, however, lends itself nicely to being implemented as a system of declarative constraints in accordance with the SHADOW paradigm.
Designing an application under the SHADOW System is a bit like directing a play. The director decomposes the desired scene into its component parts of scenery and actors. Each actor is given his or her role and behavior before the scene begins and waits for the director's cues to enter or exit the arena of the play in progress. By analogy, the SHADOW designer must dissect his or her vision of the finished application into a containment hierarchy of subsystems. The purpose and behavior of each subsystem must be defined as must the activation scope of each link.
<table>
<thead>
<tr>
<th>Element</th>
<th>Activation Time</th>
<th>Deactivation Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocks System</td>
<td>Program Start</td>
<td>Program End</td>
</tr>
<tr>
<td>Display</td>
<td>Program Start</td>
<td>Program End</td>
</tr>
<tr>
<td>Screen Layout</td>
<td>Display Ready</td>
<td>Program End</td>
</tr>
<tr>
<td>Bounding Walls</td>
<td>Screen Initialization</td>
<td>Program End</td>
</tr>
<tr>
<td>Paddle</td>
<td>Display Ready</td>
<td>Program End</td>
</tr>
<tr>
<td>Ball</td>
<td>Display Ready</td>
<td>Program End</td>
</tr>
<tr>
<td>Array of Bricks</td>
<td>Display Ready</td>
<td>Program End</td>
</tr>
<tr>
<td>Brick</td>
<td>Array Initialization</td>
<td>Impact</td>
</tr>
<tr>
<td>Motion Manager</td>
<td>Program Start</td>
<td>Program End</td>
</tr>
<tr>
<td>Collision Detector</td>
<td>Program Start</td>
<td>Program End</td>
</tr>
</tbody>
</table>

**Figure 7-14:** Elemental decomposition of the Blocks program

One possible approach to the decomposition of the Blocks application is given in Figure 7-14. By this model, the program consists of ten modules, many of which are comprised exclusively of local links (no contained subsystems). Additionally, most of the subsystems have very limited behavioral states. The bounding walls are created at a location and remain there for the duration of their existence without regard for the overall state of the application. Likewise, the bouncing ball is always a ball whether it is flying through empty space or bouncing off of other objects. It always has a motion vector associated with it and, while alterations to that vector may change the way its behavior manifests itself, it does not alter the ball's continuous, inherent behavior.

The decomposition for the Blocks program may be translated almost directly into the SHADOW Talk system level specification shown in Figure 7-15. In this design, the display is managed using the X Windows interface subsystem (recycled from the widget set exercise), the four object elements are each modelled as contained subsystems, and the two behavioral
elements (linear motion and collision detection) are addressed by applying generic constraints. A simple state diagram is used to stagger the activation times of the subsystems and to end the program.

**System: Blocks**

*Input Events: ScreenReady*

*Output Events: (not applicable to systems)*

![System Diagram]

**Figure 7-15:** The system level specification for the Blocks application
The decomposition of the application given in Figure 7-14 suggested that the basic screen layout could be addressed by modelling the screen's inverted U-shaped boundary as a composition of three individual, immobile walls. This approach allows the screen subsystem to be implemented as specified in Figure 7-16. The subsystem consists of a local output link which clears the screen when needed (as signaled by a Redraw dependency or a change in the window's geometry), a local processing link which determines the position and geometry of the individual walls, and three instances of a bounding wall subsystem. The screen never alters its dynamic behavior, and as such, has no need for individual activation flags or multiple states.

**Subsystem: BaseScreen**

**Input Events:** (none)

**Output Events:** (none)

---

**Figure 7-16:** The screen layout specification for the Blocks application
To complete the specification of the base screen, the bounding wall subsystem must also be created. This is one of the applications most primitive subsystems and is used to reserve a rectangular region of the screen as impassible for the ball and to display a visual cue to the user as to where this region lies on the screen. A specification which meets these requirements is given in Figure 7-17.

**Subsystem: Bounds**

**Input Events:** (none)

**Output Events:** (none)

![Diagram of Subsystem Bounds](image)

**Figure 7-17:** The specification of a bounding wall in the Blocks application

Like the base screen itself, the bounding wall has virtually stateless (or more correctly, uni-state) operation. Local processing is confined to a single output link which draws a rectangular box on the display according to the position and extend information it receives from the subsystem's property variables.
The unusual part of this definition lies in the existence of two additional property variables, Fixed and Shape, who have default values associated with them but serve no purpose internally. Recall that the top level specification called for object collision detection to be implemented as a generic constraint. Generic constraints are dynamically bound to subsystems based on the existence of lexically scoped properties. In this case, Fixed and Shape are two properties which will be needed later to alert the generic constraint, TimeStep, of the wall's existence.

The next subsystem required by the Blocks specification is that of the paddle. Like the bounding wall, its purpose is to declare a portion of the screen area as impassable to the ball while providing feedback to the user regarding its location. Unlike the wall, however, the paddle has a fixed geometry and its position is a function of the mouse pointer's horizontal position within a fixed range. Its SHADOW Talk specification is given in Figure 7-18.

Under this design, the paddle consists of two processing links and an output link. The Layout link uses the window geometry to constrain the possible locations of the paddle to a desired range while the Move link utilized the X coordinate of the mouse pointer's location to define the actual location. The output link is used to give visual feedback to the user. Like the specification for the bounding ball, property variables called Fixed and Shape are declared for the purpose of interfacing with the collision detection constraint.
Subsystem: Paddle
Input Events: (none)
Output Events: (none)

![Diagram of Paddle subsystem]

**Figure 7-18:** Specification of the Blocks application's Paddle

The most visually active element of the Blocks application is the Ball subsystem. Its specification is given in Figure 7-19. In addition to merely drawing the ball at its current location, this definition reserves space for recording the object’s trajectory across the screen.²⁶

²⁶The data flow graph for the Ball subsystem contains an additional processing link, Check, which has the potential to alter the ball’s position and trajectory. This link was added for demonstration purposes to reset the ball should it escape off the bottom of the screen. Most likely, if the Blocks program were intended as an actual game rather than a concept demonstration, this link would be replaced by an additional generic constraint responsible for managing balls in play and penalizing the user for lost balls.
It does not actually use this information to move itself to new location with the passage of time, however, that task is left to the linear motion constraint. Binding to said constraint, as well as the collision detection constraint is achieved through the existence of the property variable Free.

**Subsystem: Ball**

*Input Events: (none)*

*Output Events: (none)*

---

**Figure 7-19:** The Ball subsystem of the Blocks application

The final object element of the Blocks application specification is the the array of bricks. This array is actually comprised of a collection of individual bricks (a total of thirty-five initially in the example given in Figure 7-13). Since each of these bricks has the potential to be removed without the knowledge or consent of its neighbors, the master design calls for each brick to be a self contained subsystem.
The prospect of creating an array of such subsystems by explicitly connecting each and every instance to the array's data flow graph has great potential to be both tedious and convoluted, especially in the context of a graphical specification language where both physical space and visual complexity are at issue. Fortunately, the SHADOW System's support for virtual subsystems may be used to alleviate this problem. This solution is illustrated in Figure 7-20:

**Subsystem: BrickArray**

**Input Events:** (none)

**Output Events:** (none)

---

**Figure 7-20:** Specifying an array of bricks using virtual subsystems

By this definition, the BrickArray plugboard declares that all instances of Brick will be linked to the same Redraw signal variable. Issues with respect to the various positions of the bricks are deferred, making them the responsibility of the individual instances to resolve. The processing core of the BrickArray subsystem is a demand processing link which issues a series of calls to the run time engine, requesting that it allocate a given number of instances of Brick.
and that each instance should be associate with the activation flag READY. Once the bricks have been allocated, the subsystem transitions to the Ready state and the new bricks become active.

The first task each brick faces is determining its correct location in space. This is achieved through a communication channel to a trivial underlying application program. As each brick is activated, a demand processing link alerts the underlying database of its existence and request a unique identification tag, position and geometry information. This information is stored and used locally and the underlying application is not contacted again until the brick is destroyed.

The actual process of destroying a brick is triggered by receiving the signal event ImpactDetected. This event causes a state transition which activates another demand link responsible for erasing the brick, notifying the underlying application of the brick's destruction, and requesting the run time engine to deallocate the subsystem. The brick itself has no internal means for detecting a collision and relies on the generic constraint TimeStep for its trigger signal. These concepts are embodied in Figure 7-21.
Subsystem: Brick
Input Events: ImpactDetected
Output Events: (none)

Figure 7-21: A brick specification which uses an external application to self-initialize

With the exception of allocation and deallocation the subsystems in the Blocks application are essentially passive objects which simple present themselves to the user as feedback with respect to their current positions. All active behavior in this program has been deferred to the domain of two generic constraints, LinearMotion and TimeStep. The first (and simpler) of these elements is presented in Figure 7-22.
Constraint: LinearMotion
Input Events: (none)
Output Events: (none)
Filter: SHADOW_Active_System(Body)

Figure 7-22: A generic constraint implementing linear motion

The LinearMotion constraint applies to any subsystem, Body, that possesses the properties of Xreal, YReal, XPos, YPos, XVel, and YVel. Additionally, the constraint will associate the properties OldTime and Time with the subsystem if it does not already have such variable
defined. The constraint has one final restriction on its scope, that being a filter function which requires the subsystem in question to be active at the time of re-evaluation.

Internally, the constraint consists of two processing links. The first calculates the net elapsed time between the time when the object was last updated (OldTime) and its desired temporal position (Time). The second link, Move, calculates the appropriate X and Y locations, as well as their quantitized screen position equivalents, based on the object's starting position, velocity and elapsed time. These values are then used to update the various position and motion properties of the subsystem and the OldTime variable is updated to reflect the time index of the most recent update.

To function correctly, however, the LinearMotion constraint requires that the property Time be set to some value greater than to equal to OldTime. This information is essential for informing the constraint how far, temporally speaking, the object is to be moved.

Setting the Time variable is the responsibility of the collision detection constraint, TimeStep. Additionally, TimeStep is responsible for orchestrating changes in the ball's trajectory vector as a result of any collision and for notifying individual bricks of any collisions which involve them. The specification for this constraint is given in Figure 7-23.

TimeStep is designed to apply to a pair of subsystems simultaneously. The first subsystem, Body, is required to possess the properties XPos, YPos, XVel, YVel, Width, Height, Free and associates the additional properties of OldTime, Time, and Bounce if the subsystem did not already possess such. The second subsystem, Block, need only possess the core properties of XPos, YPos, Width, Height, Fixed, and Shape. At run time, any and all combinations of subsystems matching these profiles become eligible for evaluation under this constraint.
Constraint: Timestep
Input Events: (none)
Output Events: (none)
Filter: (SHADOW Active System(Body) && SHADOW Active System(Block))

Figure 7-23: A generic constraint to detect and service two body collisions
Internally, the constraint consists of a single processing link which takes position and extent information from both subsystems (one in motion and one fixed) and determines how far, up to and including the current time index, the body in motion may continue to travel before hitting the block in question. If this time window is less than the one currently recorded in the Body subsystem, the Time property is trimmed accordingly. If the time window is reduced to zero, the link assumes a collision has occurred alters the trajectory of the Body subsystem based on its angle of incidence against the Shape of the Block subsystem. Should a collision occur, the link also sends the event token ImpactDetected to the Block subsystem in question.

In practice, this system of basically passive objects and generic constraints leads to an extremely flexible development paradigm. For example, modifying the program to place additional barriers (bounding walls) inside the playing field requires the modification of exactly one subsystem and is highly isolated from the more complicated code designed for managing object motion. Likewise, the process of evolving this demonstration program into a game where each time a brick is destroyed a ball is created in its place and the player's final score is based on the number of balls left on the screen when the last brick is removed would involve very little effort beyond adding a few demand processing links and passing a token from Brick to a virtual Ball allocator.

**DEVELOPMENT OF A GESTURE RECOGNITION APPLICATION**

The final application to be presented in this section was selected to illustrate how the features of the SHADOW System may be combined to address some of the needs more commonly
associated with non-WIMP user interactions. This application, Gesture, was designed to monitor a continuous stream of input data looking for probabilistic tokens.

Two variations of the Gesture program were actually developed in the course of evaluating and applying the SHADOW System. The first was a two dimensional desktop version which recognized gestures as a function of movement of the mouse pointer. The second was a three dimensional head-mounted display version which recognized gestures as a function of the movement of a Polhemus tracker attached to the user's index finger. The conceptual basis for the gesture recognition scheme used in both programs was the same. For simplicity of presentation, only the desktop version will be discussed here.

The desktop Gesture application was designed to mimic the types of issues faced when developing pen-based interfaces or other dynamic\textsuperscript{27} gesture recognition problems. The program uses a technique of incremental recognition similar to the scheme used by Zhao [143] in developing a diagram editor. The Gesture program itself is also, nominally an editor program, though of limited scope and functionality. Initially the system presents the user with a screen containing four colored rectangles, then, by means of mouse based gestures, the user has the option of deleting, copying, and pasting rectangles anywhere in the drawing area. A small underlying application is responsible for actually creating and deleting the boxes as well as for maintaining the application clipboard's contents. The SHADOW Talk portion of the application is responsible for providing all feedback to the user with respect to changes in the underlying model, monitoring the mouse and detecting the presence and location of gesture tokens, and notifying the underlying application in the event that such a gesture is encountered. A screen dump of the finished application in operation is provided in Figure 7-24.

\textsuperscript{27}The term dynamic in this context is used to differentiate gestures which convey meaning through motion versus those, such as many finger-spelling sign language alphabets, whose primary information conduit is through attaining a given orientation and position rather than the path taken to achieve said position [120]
The actual gestures recognized by the application are defined as sequences of short strokes. A stroke, in this case, is defined by sampling the pointer's position every few pixels to create a smooth curve. Additionally, the slope and direction of any line segment connecting two adjacent points on the curve must fall within a fixed range of tolerance associated with the individual stroke. The actual length of any given stroke or the scale of the entire gesture was not considered to convey any meaningful information in this example and was ignored.
Figure 7-25: The stroke components of the three recognized gestures

The three recognized gestures are shown as a function of their base strokes in Figure 7-25. The number of strokes per gesture has little bearing on the recognition technique used, the fact that each of the cut, copy and paste gestures used by this application consisted of five strokes was simply a matter of coincidence and not a restriction on the utility of the technique.

The strokes shown in Figure 7-25 are actually idealizations of the expected pointer movement (especially given the fact that the input channel of this application was connected to a mouse rather than a digitizer tablet or a pen). In practice, the strokes were given high enough tolerances such that test users took to summarizing the gestures as: Copy, a counterclockwise ring ended in a down-left sweep; Cut, a counterclockwise ring ending in a horizontal right sweep; and, Paste, a clockwise circle ending in a down-right sweep, or, as one subject put it, "just copy in a mirror".

The system level specification for the Gesture application is given in Figure 7-26. To perform its intended task, Gesture uses a system of low-level and high-level recognizers. The low-level
recognizers operate in series within each gesture and seek to identify individual strokes. The high-level recognizers (visible in Figure 7-26 as the contained subsystems CutSign, CopySign, and PasteSign) operate in parallel with one another, simultaneously parsing a single input data stream. When any one of these links recognizes its associated gesture, it raises a discrete event token to notify the balance of the application of its discovery.

Like the other two dimensional, graphical applications presented in this section, Gesture uses the X Window interface subsystem originally developed for the widget set library and incorporates the three basic states associated with initializing, running and shutting down the I/O channel. In addition these base elements, Gesture has three other states in its augmented transition network designed to trigger demand processing links in the event that one of the three given gestures is encountered. These demand links are used to inform the underlying application of the user's edit request and the location on screen where the request was issued.

The remaining element of Figure 7-26 is the link Screen. This link binds to a subsystem whose sole purpose is to query the underlying application and redraw the screen with all active boxes shown in their appropriate locations and colors each time the underlying model changes or a redraw signal has been received from the X window interface module.

The actual process of recognizing an individual gesture is handled by three recognizer subsystems. Figures 7-27 and 7-28 present the SHADOW Talk specification for the module dedicated to detecting the Copy gesture presented in Figure 7-25. The behavior of this system is highly moded and despite the fact that the data flow graph for this subsystem consists of a total of eleven links no more than two are ever marked as active at any given point in time.
System: Gesture
Input Events: ScreenReady
Output Events: (Not Applicable to Systems)

Figure 7-26: The system level specification for the 2D Gesture application
Figure 7-27: The plugboard specification for recognizing the copy gesture
The CopySign subsystem is a high-level recognizer, that is to say that, taken as a whole, the output of this subsystem is a message indicating that a complete gesture has been recognized within the current input stream. To achieve this goal, the subsystem employs a bank of low-level recognizers, each tuned to recognize the start and end of a particular stroke. When combined with the subsystems augmented transition network the system becomes the equivalent of a discrete finite automaton designed to accept a particular grammar over the alphabet of time-space data.

The operation of the CopySign subsystem can best be understood by examining the behavior of the various stages of links as shown in the plugboard in Figure 7-27. The first link encountered by the input data, SampleMouse, is a dynamic processing link used to eliminate mouse jitter by only allowing the stroke recognizers to process mouse movements that resulted in the traversal of at least six pixels. While this restriction places an absolute minimum on the area a gesture must occupy, generating an accurate gesture in less than six pixels is a bit unrealistic and using such a filter to eliminate jitter (especially when the pointing device in question is an optomechanical mouse) greatly improves recognition rates while reducing false positives.

Any sufficiently large pointer movement will result in a change in the SampleMouse link's output. This data is fed to one of the five stroke recognizer links. The selection of which recognizer is active is a function of the state diagram. When a recognizer encounters mouse motions that equate to its associated stroke, it broadcasts an event token indicating the stroke has been found. If the motion encountered does not map to any known stroke within a given recognizer, the token NoJoy is broadcast.
Banks of LockInt subsystems are used in a demand processing mode to allow the system to capture the point on the screen where a given stroke was declared to be recognized. The LockInt subsystem itself does nothing but copy its input to its output whenever it is active. By using it in a demand processing mode, the subsystem becomes a typed data latch circuit, clocked by the setting and clearing of its activation flag. This allows the system to sequentially and selectively save portions of the mouse location data stream for quick retrieval of important data points.

Once all the key points of the gesture have been collected and the augmented transition graph has determined that a complete and valid gesture has been recognized, the local demand processing link SendToken calculates the center of the gesture and raises a CopyRequested event to the containing supersystem (in this case the Gesture system specification. As with all demand links, a DONE event is also broadcast automatically to the local subsystem.

The augmented transition network for the CopySign recognition subsystem is given in Figure 7-28 and functions as a simple grammar automaton. The system begins in the SeekOne state looking for the first stroke of the gesture. If this stroke is found, then the system proceeds to SeekTwo and so on. This continues until the entire gesture has been recognized, at which point the flag to fire off a discrete token to the containing supersystem is set and, when the associated task is complete, the system resets itself to start looking for the first stroke of the next invocation of the gesture. If at any point in the cycle the correct ‘next’ stroke is not recognized by the associated links, a signal of NoJoy is fired causing the system to reset its search from the SeekOne state.
**Figure 7-28:** The augmented transition network for recognizing the copy gesture

This approach to stroke based gesture recognition turned out to be not only successful at the application level but also very simple to work with and enhance. The initial version of this program was thrown together as a prototype in under an hour, was designed to recognize only one gesture and had roughly an 80% successful recognition rate. The two additional gestures were then designed and integrated in parallel with the first recognizer subsystem in a time span of about ten minutes each.

The success of this design and ease with which it was expanded to accommodate additional recognizer subsystems strongly suggest that this technique could be extended to accept a much larger vocabulary of gestures. One restriction on such extension should be noted however.
Under this scheme, no gesture should contain another complete gesture as a subset of its stroke definition. Should such a design flaw occur, the parallel nature of the SHADOW run time engine would allow the embedded gesture to fire a recognition token while still in the process of recognizing the larger one. Fortunately, this issue can be mitigated (or avoided entirely) with a little forethought and planning on the part of the gesture vocabulary designer.

CLOSING COMMENTS

The applications presented in this section represent only a small fraction of the applications, exercises, and prototypes that were created to test the claims of the SHADOW System both to operate correctly and to provide a new specification paradigm which is well suited to meet the needs of designers as interfaces migrate into the domain of non-WIMP interactions. The programs detailed here were restricted to relatively small applications which, individually, only exploited two or three features of the UIMS to complete its task. By and large, these samples had stronger ties to WIMP-style interactions than to true non-WIMP interfaces. This was done as a deliberate attempt to scope the narrative to a handful of very specific topics at a time and so that the successful operation of each language feature or concept could be demonstrated in an isolated forum without overwhelming the reader with complex, convoluted and obscure examples.

The selected examples, as well as the dozens of other efforts that went into the proofing of the UIMS, have shown that the SHADOW System works as designed at the feature level.
Additionally, development experience with programs such as the gesture recognition applications and simple physics simulations have shown that SHADOW Talk specifications are easy to work with, maintain, reuse and evolve. The one claim that has yet to be verified, however, is one of the most significant. This is the claim that the SHADOW paradigm is suitable for the specification of complete virtual worlds and that the SHADOW UIMS is capable of producing robust and responsive executable code from high level specifications. Verifying this claim by example will be the subject of the next section wherein a complete, interactive virtual world developed using the SHADOW System is presented.
SECTION VIII: DESIGN OF A VIRTUAL WORLD

OVERVIEW

The programming examples given in Section VII were each designed to test a limited subset of the SHADOW System's purported features and capabilities. These small applications, while echoing the themes commonly encountered in the creation of a large scale non-WIMP interface such as those encountered in virtual reality applications, were not required to face challenges on the same order of magnitude as those encountered in a real life, full scale development effort. Statements and projections about scalability and utility of the SHADOW paradigm may be hazarded based upon the lessons learned from these examples, but only by applying the SHADOW System to significant development effort can the claim of suitability to programming in the large be substantiated.

To complete the argument in favor of the SHADOW paradigm this section presents, in its entirety, the design and implementation of a virtual world. The work was done on a Silicon Graphics workstation using Performer as the graphics rendering engine. Two variations on the basic design were desired, the first being a desktop application which allowed the user to explore and interact with the world using keyboard and mouse controls, and the second being a head mounted display version which employed Polhemus trackers to sense the user's position, perspective, and activities within the virtual environment.

28 Although the Performer package provides a large variety of features for maintaining an underlying model such as bounding volumes and object intersection tests, none of these features were employed in this effort. Performer was used in the exclusive capacity of a three dimensional rendering package, placing the burden of collision detection and world model management on the application itself as a further test of the SHADOW System.
The application itself is called the Rookery. As the name implies, the context of the virtual world is an ice sheet populated by a couple dozen virtual penguins. The world exhibits a very high level of parallelism in its underlying processes (penguins and their toys in motion, icebergs drifting by, etc.) and, like their real world counterparts, the virtual penguins rarely sit still for very long and tend to go about their business with or without human intervention. All objects within the world have mass associated with them and are constrained by the laws of gravity. Thus, while penguins cannot fly, the virtual penguins of the Rookery can be picked up and tossed (though based on their behavior before, during and after this procedure, they do not seem to enjoy the experience). Other consequences of the physics model include the ability to stack objects upon one another and by picking up the bottom object, indirectly lift the entire column, the ability to control the range of a toss by altering an object's initial momentum and initial angle of ascent, and the ability to weigh objects in a two pan balance to determine the ratios of their associated masses.

**DECOMPOSING THE WORLD**

In Section VII, the process of designing an application under the SHADOW System was likened to directing a play. This is an appropriate analogy to apply to the Rookery application effort at a philosophical level in that it provides both a level of abstraction and a sense of context from which design decisions may be made and implemented. Like most major motion pictures to come out of Hollywood in the recent past, this play has no plot. It is a visual experience with which the user may *interact* but not entirely *control*. It has a dynamic setting and an active and adaptive cast. The cast members have both scripted behaviors and
motivations and each member is responsible for interacting with the world in its own way and in concert with the actions of others.

The first step in directing the Rookery is to establish the setting. This defines the foreground, background and laws by which the world is governed. In this particular case, the base scene consists of an ocean, a set of icebergs whose positions change with respect to time, and an ice sheet capable of supporting the weight of the cast. In addition to the base scene, the foreground also contains a two pan balance which is driven by the force of gravity and as such, the law of gravity itself completes the setting.

Figure 8-1: A scene from the Rookery application

The cast is made up of basic actors who act and interact in a variety of ways. At an individual level, the cast is made up of penguins, balls, weights, and roving teapots. Some actors
represent lone characters while others are ensemble characters and form small troupes, such as a trio of penguins who act in concert with one another or a penguin who plays with a particular ball. Collectively, this cast of characters is used to populate the world.

The final aspect to consider in the design of the Rookery is the vantage point from which the user will witness this play and the means by which he or she will interact with it. In the case of the Rookery, two alternatives will be offered the first being a desktop version and the second being a head mounted equivalent. While this decision affects how the user is to be represented in the world and how, at a lexical level, the user controls this representation, it has no bearing on how the setting itself or how the cast should behave and can safely be deferred to the end.

To establish a sense of context, Figure 8-1 illustrates a sample scene from the completed Rookery application. Three weights can be seen in the immediate foreground and a fourth can be found in the left hand pan of the two pan balance. Behind the balance is a penguin who seems to have taken an interest in the device; to his right is a ball sitting idly on the snow; and behind him in the midground can be seen another penguin walking away from a second ball. In the background, an iceberg drifts by in the ocean.

**ESTABLISHING THE SETTING**

The program specification calls for a setting consisting of: an ocean, which has no special behavior associated with it and is essentially a visual placeholder to complete the illusion of the world; an ice sheet, which constitutes the stage upon which the Rookery play is to be
performed and must be able to support the cast of character placed upon it; a two-pan balance which reacts to changes in its environment; and a collection of icebergs drifting in the distant ocean whose position is simply dictated by the passage of time.

Figure 8-2 shows a bird’s eye view of the world of the Rookery. The rectangular box in the center of the diagram represents the placement of the ice sheet itself while the four small boxes near the edges represent the drifting icebergs. The dark rings associated with each iceberg depict the paths through which the objects orbit the main sheet while the arrows within each box represent the direction of rotation. Circular paths for the drifting ice may be a bit unrealistic by real world standards but within the context of a virtual world and limited field of vision, these orbits, combined with random speeds of rotation, provide a quick and simple means to create the illusion of far-field motion.

Translating this description into a SHADOW subsystem can be achieved with very little effort. The first step in the process involves deciding on how to decompose the scene into its base elements. Much of this task, however, can be derived directly from the description. The two pan balance has properties and behaviors which are distinctly different from the rest of the setting and as such, should be made into its own subsystem. Likewise, the passage of time (needed for the animation of the icebergs) is a dynamic input which is independent of its use in this module and thus, should be the product of another subsystem, not a local production. The ocean and icebergs share a common activation scope and behavior, namely the ability to exist at a mathematically predictable space at any given time regardless of user involvement and may be specified and serviced collectively.
Figure 8-2: A top view of the scene layout for the Rookery

Figure 8-3 shows a SHADOW Talk specification for the basic scene based on these decomposition decisions. The subsystem has two states, an initialization state which is entered when the module is activated for the very first time, and a run state which represents the normal operating mode for the subsystem for the duration of its lifetime.
Subsystem: Scene
Input Events: SHADOW FrameComplete
Output Events: iceReady

Figure 8-3: The specification of the basic scene

The data flow graph consists of four links, two of which are contained subsystems, one to read the current time index and the other to create and manage the two pan balance. The remaining two links perform local initialization and generate feedback to the user.

The demand processing link, AddSetting is evaluated exactly once upon the occasion of the subsystem's first activation. This link is responsible for instantiating the graphical objects (defined in external C++ files) that will be used to represent the various aspects of the scene.
Next, this link must register the existence of the primary ice sheet with the underlying application\textsuperscript{29} so that it will be available to support the weight of the cast when they are eventually created. When this task is completed, the link raises the event IceReady so the balance of the world becomes aware that the stage is set.

The final link, Draw, is responsible for repositioning the icebergs with respect to time. Under the Performer package, which maintains its own scene graphs and refresh rate timing, the process of animating motion does not directly involve the programmer dictating which pixels should be lit. Instead, all that is required is that the position of the object inside the scene graph be updated, leaving the actual rendering issues to the Performer engine.

The careful reader will note that the subsystem specified in Figure 8-3 had the event SHADOW\_FrameComplete listed as a desired input but the associated augmented transition network did not make use of this token. The purpose of this entry in the event mask at this point is to allow the event (which initiates in the run time engine) to pass through this module and enter the Balance subsystem, where it will be processed accordingly.

The balance itself consists of three moving parts, the two pans and the balance bar which spans between them. The function of the balance is to adjust the height of the two pans and the angle of the bar between them as a function of the ratio of masses being supported by the two pans. This is achieved through the specification given in Figure 8-4.

\textsuperscript{29}The term application is a bit generous for the amount of functionality found in this external system. In essence, the underlying application of the Rookery consists of a poor man's database where elements may register themselves and query one another for proximity.
Subsystem: Balance
Input Events: SHADOW_FrameComplete
Output Events: (none)

Figure 8-4: Specification of a two pan balance

Just as in the Scene subsystem (and indeed all subsystems which involve one-time memory allocation) the balance begins in an initialization state where the graphical objects representing portions of the model are instantiated and added to the Performer scene graph. Once these
bookkeeping tasks are complete, the subsystem enters its normal operating state where the
dynamic processing link, Ratio, adjusts the height of the two pans based on the masses being
supported by each and the output link, Draw, uses these heights to reposition the graphical
avatars within the scene graph.

Once the masses in the two pans is known, the process of calculating the position of the pans is
no more difficult than tracking the paddle to the mouse position in the Blocks application. The
trick to managing the functioning balance is having a scheme that will allow the new masses to
be found quickly. This is the objective of the state, CalcState. This state can be entered only in
response to the event SHADOW_FrameComplete. This is a signal sent by the run time engine
to the main system definition to alert it that it has completed drawing an output frame and that
time is available for supplemental processing before the next output cycle begins. In this space
between frames (potentially up to a fifteenth of a second, depending on rendering complexity
and the run time configuration of Performer) the subsystem activates the demand processing
link, Recalc, which queries the underlying model for the net mass of all objects being directly
or indirectly supported by each pan and produces a snapshot of this information for use in
rendering the next frame of output.

The other subsystem referenced by the Scene specification is the TimeIndex module. Designed
as a generic input device interface, the TimeIndex subsystem simply queries the SHADOW run
time engine for the current value of the engine's internal counter\(^{30}\). This subsystem is used
extensively through the design of the Rookery whenever real time animation is required. Its
annotated SHADOW Talk specification is given in Figure 8-5.

\(^{30}\)The time index is initialized to zero at the start of a programing execution and is incremented in real time
once every 10 milliseconds give or take the resolution of the target platform's real time clock.
Subsystem: TimeIndex
Input Events: (none)
Output Events: (none)

This system queries the SHADOW run time engine for the current time index, a long word counter which is initially at zero when the program begins and is updated in real time at the resolution of the system gettimeofday function. This information is useful for timestamps, physics simulations, and for forcing updates of non-pollled links as part of a dependency thread.

Figure 8-5: The TimeIndex utility subsystem

As stated in the overall description of the design, the laws of physics are every bit as much a defining characteristic of the penguin world as icebergs or oceans and should be addressed early on in the design. The laws of physics can be very broad and complex, for the sake of simplicity and practicality, the Rookery reduces these rules to a handful of physics-based properties and assumptions. These are: time is a strictly increasing function; unsupported objects fall with linear acceleration; an object is supported if the object immediately below it is supported; and, motion perpendicular to the force of gravity is immune to its effects. In addition, four special rules were added which have no basis in physics but result in convenient
simplifications: all cast members within the virtual world are solid and non-compressible; the ice sheet is self supporting; the user is exempt from physical laws; and, objects held by the user are exempt from gravity and may be placed anywhere in the world.

**Constraint: ProjMotion**

*Input Events: (none)*

*Output Events: (none)*

*Filter: SHADOW_Active_System(Body)*

---

**Figure 8-6:** The physics manager for the Rookery application

This list of assumptions and rules resulted in the creation of a generic constraint similar to the one use by the Blocks application to control the motion of the ball. This module, ProjMotion, is shown in Figure 8-6. This constraint is scoped to bind to any subsystem which has the
properties of position, velocity and an item identification tag signifying that it is a part of the underlying model. Any subsystem which meets this criteria will have an additional property associated with it, OldTime, to allow the constraint to keep track of the amount of time elapsed between updates of any given body.

Unlike the Blocks application which used two generic constraints operating in concert, one to move objects and one to detect and service collisions, the Rookery employs only one module which in turn queries the underlying model regarding mobility and the overlap of bounding spaces\textsuperscript{31}. Elastic collisions such as those found in the Blocks program are not incorporated.

**DEFINING THE BASIC ACTORS**

. Once the setting for the Rookery has been established, the design process turns to one of creating the cast that will populate the ice sheet. There are four primitive entities in the penguin world which will be referred to as basic actors. These base classes are: Ball, Weight, Teapot, and Penguin\textsuperscript{32}. The level of complexity of each of these entities ranges from the module Weight, whose sole purpose is to take up space and have mass, to the articulated, animated and self-motivated entity of the penguin actors.

All actors, regardless of class share certain properties and associated behaviors. They are all registered elements of the underlying model. They all have associated masses and bounding

\textsuperscript{31}The use of the model in this capacity was done as a matter of choice, not necessity, and to present an alternative approach to the use of generic constraints compared to those already described in the previous section.  

\textsuperscript{32}To some it may seem unusual to think of a ball or a weight as an actor (the teapot and penguin are self motivated and are easier to anthropomorphize). The term actor is used in these cases to differentiate these entities from the more generic term object (which could potentially be referring to any one of dozens of aspects of the Rookery application) without introducing extraneous terms such as prop.
volumes (this information is stored externally in the world model). They have locality, motion vectors and the potential to be picked up, carried, and/or tossed by the user. These commonalities may be addressed in a variety of ways.

The simplest way of dealing with a common behavior across multiple subsystems is to ignore it. Each subsystem is implemented as if its functionality were entirely unique and essentially redundant code is introduced into the final application. The price for this naive solution is a larger, less maintainable system filled with re-invented wheels and a development cost which reflects the duplicate efforts.

A better solution is to modularize the commonalities into shared and reusable components. Creating generic constraints, such as the one used by the Rookery application to enforce physical laws, is one way in which this can be achieved. This has the advantages of centralizing the specification and maintenance of a desired commonality so that the behavior may be shared and reused by as many differing classes of subsystems as needed. The disadvantage of this approach is marginal: generic constraints are scoped based on the lexical names given to its associate properties and are dynamically bound, depending on the size and complexity of the overall application, the implicit association between the generic constraint and its target subsystems may not seem obvious to the system's maintainers.

An alternative way of specifying a common behavior is to modularize it into a generic subsystem which is explicitly embedded in any subsystem which desires to share the functionality. This approach offers the same advantages of centralizing the specification and maintenance while avoiding the potential confusion associated with implicit behaviors. The
disadvantage of this scheme is that the inclusion of the generic subsystem in the target modules adds to the visual complexity and physical space requirements of the individual plugboard specifications.

**Subsystem: Holdable**
*Input Events: Gotcha, Released*
*Output Events: (none)*

![Diagram of Holdable System](image)

**Figure 8-7:** A generic subsystem which enables its container to be picked up and carried

To illustrate this third approach, a generic subsystem can be created to service the actors' common behavioral characteristic of being capable of being held by the user. Such a subsystem is depicted in Figure 8-7. The module has two states: FreeState, the default, in which the subsystem does essentially nothing; and, HeldState, in which the subsystem activates another generic subsystem to bind the position and velocity vector of the user's hand.
to the position and velocity vector of the module. Transitions from one state to the next are achieved by reacting to the signals Gotcha and Released which originate in the subsystem controlling the user's hand on the occasion of a successful grab attempt or release of a given object.

In addition to controlling the activation of the GrabLock subsystem, the Holdable subsystem also maintains a pair of demand processing links, one of which is evaluated each time the module changes state. These links are used to notify the underlying application of when the object is being held by the user. This state information is important in the generic constraint ProjMotion will eventually query the model and needs to know whether or not the object should be temporarily exempted from the physical laws which govern the Rookery.

To function properly, Holdable requires the services of the subsystem GrabLock. This simple subsystem is shown in Figure 8-8. It, too, is a two state module but unlike Holdable it does not toggle between states within its activation scope. Instead, GrabLock has a universal initial state. This is to say that each and every time this subsystem is activated the augmented transition network and its associated flag values reset to the conditions defined in the state InitState. This is done to force the demand processing link FindOffset link to be evaluated exactly once each time a grab is initiated. FindOffset is used internally to calculate the spacial offset between the origin of the hand's avatar and the origin of the object's. This information is used by the dynamic processing link Follow as it updates the object's position and velocity vector in response to the motion of the user's hand.
Subsystem: GrabLock
Input Events: (none)
Output Events: (none)

Figure 8-8: A generic subsystem to map the characteristics of the user's hand to an object

With Holdable and Grablock defined, they may be explicitly embedded into the specifications for each of the four actor classes. By including these subsystems and binding the actors' position and velocity properties to those of the subsystem Holdable, the actor subsystems automatically extend their innate behavior to include that of Holdable and GrabLock.
Subsystem: Ball
Input Events: Gotcha, Released
Output Events: SphereReady

Figure 8-9: The specification of a Ball subsystem

With this foundation established, the design of the four basic actor subsystems may continue, beginning with the two simplest actors, Ball and Weight. From a behavioral standpoint these two subsystems are identical, only their shapes and graphical representations differ. Each subsystem consists of two states: an initialization state and an operation state. When initializing, the subsystems each use a demand processing link to register their respective
existences with both the underlying model and the Performer scene graph. When in the
operational state, they each maintain an output link to ensure that the scene graph and the model
are synchronized and each contain a copy of the subsystem Holdable to ensure that any
attempts by the user to reposition the actors will be both recognized and accepted. The
completed specification for the Ball subsystem is given in Figure 8-9 and that of the Weight
subsystem is shown in Figure 8-10.
Subsystem: Weight
Input Events: Gotcha, Released
Output Events: WeightReady

Figure 8-10: The specification of a Weight subsystem

The third base actor class is that of the Teapot. Unlike ball and weights, which passively sit in place until disturbed by external forces, teapots are self-motivated characters inside the virtual world of the Rookery. Left alone, a teapot will pick a direction and slowly inch along in a straight line until its path is blocked at which point it will pick another direction at random and attempt to continue. Like all actors, the teapot may be picked up and relocated at any time by the user.
Subsystem: Teapot
Input Events: Gotcha, Released
Output Events: TeapotReady

Figure 8-11: The specification of a Teapot subsystem

A SHADOW Talk specification for a Teapot actor is given in Figure 8-11. This subsystem is similar in structure to that of Ball and Weight but contains one additional demand output link, Move, and an additional state, HeldState. The output link is used to manage the teapot's motion when it is free to roam about the ice. The additional state is used to prevent this link
from being evaluated while the teapot is being held by the user. As before a demand processing link is used to perform one time initialization and a Holdable subsystem is used to allow interaction with the user's hand.

**CREATING A COMPLEX, SELF-MOTIVATED ACTOR**

By far the most complicated of the base actor classes is the penguin. Unlike the other actors, which are solid shapes whose only degrees of freedom are their position in space, the penguin is an articulated entity, capable of a range of animations and poses. Figure 8-12 illustrates the various articulation points of the graphical object associated with the penguin actor.

---

**Figure 8-12:** The articulation capabilities of the penguin graphical object
Beyond the ability to simply strike a more detailed pose than a teapot or a weight, the penguin subsystem is also capable of selecting its behavior from a range of options and changing which behavior it chooses to pursue at any given point in time based on its perceptions of the world around it. Additionally, the design of the penguin allows for special property variables to be passed to the penguin actors, collectively or individually, allowing their reaction to situations and their internal decision process to be biased using high level behavioral control tags.

The penguins in the Rookery application never strike and hold a static pose, some portion of their articulated bodies is always in motion. Even when standing in one spot, the penguins will fidget, twist, shiver, raise their flippers and/or cock their heads from side to side. In addition to demonstrating how easily concurrent animation can be managed inside the SHADOW System, this constantly active behavior stream goes a long way towards supporting the illusion of a virtual world rather than a computer program with 3-D graphics.

The SHADOW Talk specification for the Penguin subsystem is given in Figures 8-13 and 8-14. The creation of this design began by defining the the subsystem's formal interface to the outside world. Like all actors, the subsystem listens for Gotcha and Released events sent from the the subsystem servicing the user's hand and issues a PenguinReady event once the subsystem's local data has been allocated and is ready for operation. The subsystem has four input variables: HandPos, a continuous track of the user's hand's position in virtual space; HandVel, the most recent motion vector of the the user's hand; Wild, a high level behavioral control affecting how dramatically a penguin reacts to being picked up; and, Wunderlust, another behavioral control affecting the likelihood that a penguin will get bored enough with its
current activity to start wandering around. Additionally, the penguin actor has nine properties explicitly associated with it: Pos, the penguin's position in virtual space; Vel, its velocity when being held, tossed, or freely falling; Orient, the base orientation of the penguin's body; Held, a flag indicating the bird is currently being held by the user; ItemID, the penguin's registration tag assigned to it by the underlying application; IPos, the position of some item in space that the penguin is likely to take an interest in; Obsessed, another behavioral control governing how important the item of interest is to the penguin; Dest, the penguin's desired destination when wandering around the world; and, Speed, a final behavioral control governing the penguin's gait as it wanders from place to place.
Subsystem: Penguin
Input Events: Gotcha, Released
Output Events: PenguinReady

Figure 8.13: The plugboard specification for the Penguin Subsystem
Figure 8-14: The augmented transition network for the Penguin subsystem

The data flow graph for the Penguin subsystem given in Figure 8-13 illustrates how these interface parameters are passed to lower level links and subsystem for actual processing. Like all actor subsystems, Penguin uses a demand processing link to deal with the issue of initialization and memory allocation. The subsystem Holdable is included to address basic behavior of relocation by the user and, again, a dynamic output link is used to synchronize the model with the Performer scene graph. The remainder of the data flow graph is dedicated to defining the penguin's high level behavior and passing behavioral control information to lower level subsystems which deal with detailed issues such as animation and timing.
The individual role of each of these subsystems can be best understood by analyzing the augmented transition network given in Figure 8-14. Beyond the initialization state common to all actor subsystems, Penguin has been defined with four operational states which correspond to the macro-level behaviors of the actor. These four operational modes are: Bored, in which the penguin fidgets in place and watches its surroundings; Fascinated, in which the penguin has noticed an item of interest within its field of vision and is enraptured with watching the item; Walking, in which the penguin is taking the necessary steps to move to a desired destination; and, and, Panicked, in which the penguin is reacting to being held by the user.

By this technique, the Penguin subsystem need only focus on the issue of when a particular behavior should manifest itself, not how the behavior is embodied. Discrete event signals are used to transition the subsystem from one operational mode to the next within the parameters specified by the augmented transition network. Some of these signals occur asynchronously and externally such as the Gotcha event which allows the user to grab any penguin at any time. Others are triggered randomly, based on the penguin’s current activity and the settings of its governing behavioral controls. This is the case for the signal PenquinMove, which can only occur if the penguin is bored and whose probability of occurrence is dictated by the values of the variables Wunderlust and Obsessed and the amount of time the penguin has remained in the Bcred state. Still other event signals require a combination of external and internal stimuli to precipitate them. One such event, NoticedObject, requires that an item of interest enter the neighborhood of the bird and that the penguin be looking in the proper direction to allow the item to be in its field of vision.
The benefit of this architecture is that it allows the overall behavior of the actor to be specified at a macro-level while deferring low level details to dedicated controllers. These controlling subsystems take on the responsibility of defining how the desired behavior will manifest within the actor's ability to articulate and how this articulation should be accomplished to produce a visually plausible animation sequence.

The simplest of these controllers manages the penguin's actions while fascinated by an object of interest. This subsystem is shown in Figure 8-15. The locations of both the penguin and its associated point of interest as well as the penguin's orientation within the world are used by the subsystem to determine if the object is both close enough and visible enough to alter the penguin's reaction to it (NEAR) or too far away to be of active interest (FAR). This decision is made by using two dynamic processing links which are temporally disjoint in activation scope. If the object is too far away to be of note, the link Detect passively watches for its approach and raises a NoticedObject signal when conditions warrant. If the object has already been noticed, the link Watch monitors its motion, calculates articulation information for the penguin's flippers and, eventually raises an OutOfSight signal if the object moves too far away or beyond the bird's ability to crane its neck. In addition, the subsystem employs an output link, Stare, which is only active while the penguin is active watching the given object. This link is used to articulate the actor's flippers and head to coordinate the bird's focus vector with the position of the object in question.
Subsystem: PenFascinate
Input Events: (none)
Output Events: OutOfSight, NoticedObject

![Diagram of PenFascinate subsystem]

**Figure 8-15:** A subsystem to manifest a penguin's fascination

The PenFascinate subsystem gains much of its simplicity from the fact that its operation, while slightly more elaborate graphically, is conceptually identical to the techniques used to bind the paddle to the mouse in the Blocks application and the mouse based control of the slider handled in the widget set example. The penguin simply achieves a static pose that is a direct function of the current location of the object of interest. The behaviors embodied by the penguin's idle and panicked states are slightly more complicated. In these cases, the penguin is no longer trying to achieve a static pose but is randomly articulating itself within parametric boundary and behavioral constraints.
**Subsystem: Penidle**

- **Input Events:** (none)
- **Output Events:** (none)

---

**Figure 8-16:** A subsystem to control the activity of an idle penguin.

The controlling logic for managing an idle penguin is given in Figure 8-16. In the world of the Rookery, idle behavior for a penguin consists of moving its head from side to side as well as
up and down to scan its surroundings for an item of interest. Concurrently with this, the penguin will independently raise or lower its flippers and fidget in place. These movements are managed by the demand processing links Scan and Fidget. Rather than defining these motions as direct mappings of trigonometric or hyperbolic functions\(^3\), these links control the motion of each appendage by establishing a general movement trend and, with the passage of time, randomly determine whether to continue to follow the given trend, momentarily pause while maintaining the trend, or to establish a new trend with the the limits of the degrees of freedom associated with the actual model. As with the fascination subsystem, a dynamic output link is used to convert the individual articulation decisions into an overall animation frame in the Performer scene graph.

This same structure is used to specify the penguin's panic reaction, as illustrated in Figure 8-17. The panicked behavior is also visually similar to the idle behavior in that it, too, consists of panning the head from side to side and randomly flapping the flippers. Additionally, since the panic reaction only occurs when the creature is being held, the animation also includes foot motions. As in the case of the idle activities, two demand processing links working in conjunction with motion tendency variables are used to determine the desired level of articulation from frame to frame.

The internal operations of these links are slightly more complicated than their counterparts in the PenIdle subsystem due to the existence of the behavioral control variable Wild. This value is used to govern how violently the penguin reacts to being picked up and held. While the individual articulations are still random functions, the setting of the Wild input is capable of skewing the probabilities either for or against rapid thrashing motions.

\(^3\)Experimentation with this approach was found to yield movements that seemed robotic and unnatural to the human observer and were detrimental to the overall illusion of the Rookery.
Subsystem: PenPanic
Input Events: (none)
Output Events: (none)

Figure 8-17: A subsystem to manage the panic reaction of a penguin

Just as in the PenIdle example, PenPanic uses a dynamic output link to perform the actual frame to frame animation of the penguin as it thrashes about in panic mode. In this particular
case, the desired head and flipper positions as fed to the link directly while the the positions of the feet are calculated internally as functions of the positions of the other appendages.

Both PenIdle and PenPanic possess only one operating state each. This is a consequence of the hierarchy of specifications. When these subsystems are active they become responsible for governing how their respective behaviors are to manifest and for orchestrating the animation of the associated graphical objects. The decision as to when a particular behavior should or should not be exhibited is beyond the scope of these subsystem and is left to higher authorities.

The final behavioral mode for the penguin is also the most complex. This is the subsystem which controls the penguin's ability to walk in a visually plausible\(^{34}\) manner. The design of this subsystem, shown in Figure 8-18, decomposes the task of walking to a given destination to literally one step at a time.

The basic animation and locomotion concept is similar to an approach used by Mansk and Muhlauser to animate two dimensional comic actors using matchable sequences of static frames and transition diagrams [97]. The subsystem consists of three states: Ready, in which the penguin has reached a point in its gait where it must decide what action to take next; Left, where the penguin is actively pursuing a stepping action with its left foot\(^{35}\); and, Right, where the penguin is actively pursuing a stepping action with its right.

\(^{34}\) An assortment of techniques for synthesizing walking motions in aminations has been explored including those described in [6], [88] and [105] and while the specification given in this example bases its animations on photographic examples of actual penguins, nothing about this technique precludes the incorporation of computer synthesized, precalculated motion sequences in place of manually generated animations.

\(^{35}\) This is a slight misnomer in that, as with any creature subject to active balance restrictions, the penguin "steps" using its entire body to avoid tipping over. The designations of left and right foot are simply a convenient notation to indicate which foot is in contact with the ground and which is in the air at any given point in time.
Subsystem: PenWalk
Input Events: (none)
Output Events: DestinationReached, DestinationBlocked

Figure 8-18: A subsystem to animate the waddling motion of a penguin
The three states are directly coupled to the activation of three dynamic output links. The links StepRight and StepLeft, as the names imply, are used to orchestrate the positioning and time based animation of the penguin actor as it performs a single step to the right or left. The link MovePenguin serves as the local dispatcher for the two possible animation tasks. This link compares the penguin's desired destination with its current bearing and, in conjunction with the penguin's history of previous steps determine which foot should be the next to move. This allows the penguin to turn gradually as it walks and follow a curving path to its destination rather than performing a pirouette in place followed by straight line navigation.

The four motion controller subsystems nested within a higher level behavioral automaton allow the designer to segregate animation and timing issues from the design of individual activities and individual activities from the specification of overall behavior. In addition, the existence of variable parameters which govern the macro-level behaviors allow the penguin actors to be given high level directions such as "go over there" or "watch this" without burdening the application designer with the mechanics of how each task will be accomplished internally by the penguin. This feature is exploited extensively in the design of the ensemble acting troupes.

**DEFINING ENSEMBLE TROUPES**

The basic actor subsystems of Ball, Weight, Teapot and Penguin represent the primitive character types which populate the virtual world of the Rookery. While these characters may be briefly interesting, they possess little, if any, agenda of their own and filling a stage with actors without purpose achieves little. To compensate, the cast of the Rookery includes
ensemble troupes. These are collections of actors whose behaviors play off of one another to maintain a certain momentum of the action within the Rookery regardless of the user's involvement.

There are three classes of ensemble troupes in the Rookery’s design. The first, nicknamed Pele, consists of a penguin who is obsessed with a ball. The second is the user’s fan club, which consists of a trio of penguins who have a tendency to follow the user around as he or she explores the virtual world. Lastly, the third class of troupe is a trio of penguins who are obsessed with chasing down a roving teapot and trapping it between their bodies to prevent it from escaping.

The penguin actors were designed to respond to high level behavioral directives, this greatly simplifies the task of creating and controlling ensemble troupes. As a first example, the specification of the Pele penguin subsystem is given in Figure 8-19.

In this design, the location of a ball actor, BPos, is coupled to the penguin's point of interest, IPos. Additionally, an explicit local variable is used to set the penguin's behavioral control property Obsessed to a value which indicates a high level of fixation on the idea of watching and maintaining close proximity to the object of interest (in this case, the ball). Other behavior-related variables, such as the penguin's movement rate or tendency to panic when held are coupled to the Pele subsystem's own interface so that they may be tuned at higher levels if desired.
**Subsystem: Pele**

**Input Events:** (none)

**Output Events:** (none)

---

*Figure 8-19:* A subsystem designed to encourage a penguin to chase a ball

This same technique of coupling basic actors together to synthesize higher orders of behavior is repeated and extended in the design of the other two ensemble troupes. Both of these ensembles include a cluster of three. Keeping with the theme of modular development the trio of actors may be clustered together and used as a building block for completing the ensembles.
Figure 8-20: Coupling three penguins to play follow-the-leader

Figure 8-20 depicts a SHADOW Talk specification for a subsystem which strongly encourages three penguin actors to play a game of follow-the-leader. According to this definition, one penguin is designated as the leader and its position (adjusted by a slight offset to discourage the
penguins from bumping into one another) is used as the dynamic point of interest for the other two actors. As with the case of the Pele subsystem, the Obsessed control for each of the non-leader penguins has been set to a value which strongly encourages the actors to pursue their points of interest. The point of interest and level of obsession for the leader is coupled to the Triad subsystems own interface to allow higher levels of behavioral abstraction.

**Subsystem: FanClub**
*Input Events: (none)*
*Output Events: FanClubReady*

![Diagram of FanClub subsystem]

**Figure 8-21:** Specification of the user's penguin fan club
The first example of such an abstraction can be seen in Figure 8-21. This is the specification of the user's penguin fan club. The club consists of a trio of penguins who make a habit of following the user as he or she explores the virtual world of the Rookery. This behavior is relatively simple to achieve given the infrastructure that has already been specified.

According to the design given in Figure 8-21, the fan club consists of a Triad subsystem whose dynamic point of interest is the current location of the user's hand and, as before, the property governing the Triad's level of obsession is locally set to a high value. Tracing this logic through the various specification levels resulting in the following behavior chain: the lead penguin of the Triad is obsessed with the user's hand and actively seeks to be in close proximity to it; the other two penguins of the Triad are obsessed with their leader and actively seek to be near him; a penguin desiring to be at a different location within the plane of the ice sheet can alter its position by walking; walking is a process of alternating left and right steps while turning hips and ankles to effect directional changes; a step is a time sensitive sequence of body adjustments designed to transfer the creature's weight exclusively onto one foot while maintaining balance and positioning the other foot to advance the body. The ability of the SHADOW System to maintain continuous relationships and conduct parallel/concurrent processing means that a movement of the user's hand is directly reflected in the feet of the penguins of the fan club as they waddle about pursuing their assorted objects of obsession.

The behavior of the final ensemble troupe, the tea chasers, is conceptually a combination of the behavior of the Pele subsystem with that of the fan club. The Triad subsystem is reused to provide the tea chaser troupe with the trio of penguins it requires, but, unlike the fan club, the
user is not directly involved. Just as the Pele penguin was obsessed with a specific Ball actor, so the lead penguin of the Triad is obsessed with a Teapot actor. Although different subsystem are involved, the task is no more challenging than the process used in the other two ensembles.

The only complication to this scheme comes from the fact that the teapot actor is itself self-motivated and changes its position in space whenever possible. The design of the tea chaser ensemble, given in Figure 8-22, recognizes this fact and actually uses it to play the obsessive penguins' behavior against the teapots innate path seeking activities. In practice this results in a subsystem where, if the individual actors are initially placed far enough apart, as the teapot moves, the penguins converge on it and one another. If a single penguin gets in the path of the teapot it will reverse direction to escape the obstruction. If one or both of the remaining penguins in the Triad blocks the teapot's new path there is a high probability that the teapot will become corralled by the penguins and immobilized until one of the blocking penguins gets bored enough to wander away or until the troupe is disturbed in some way by the user.

This point can be illustrated by the screen capture of two tea chaser penguins and their associated teapot is given in Figure 8-23. At the time of the snapshot, the teapot was in motion, moving toward the feet of the forward penguin. If the remaining penguin of the Triad arrives on the scene from the right in the near future the teapot will most likely become trapped between the birds and the penguins, having successfully hunted down their prey, will slip into idle mode or fascination mode depending on their vantage point with respect to the teapot's location.
Subsystem: TeaChasers
Input Events: (none)
Output Events: TeaChaserReady

Figure 8-22: Specification of the tea chaser troupe
Figure 8.23: A pair of tea chaser penguins and their quarry
DIRECTING A CAST OF VIRTUAL ACTORS

Once the subsystems defining the basic and ensemble cast members have been established, the virtual world must be populated with instances of these actor modules. From a behavioral standpoint, the internal specifications of the actors make them fairly autonomous once created, but each actor still requires some level of stage direction. This is to say that a director module is needed to encapsulate and initialize each actor subsystem and to define and manage the activation scopes (in this case the entrance and exit cues for each actor) for these subsystems.

Figure 8-24 illustrates the master casting director for the Rookery application. For simplicity, all actors in the world share the same activation scope, that being the life of the program. In the final design, the world is populated by four weights, one ball, one lone penguin, one fan club, one Pele penguin, and two sets of tea chasers for a total of nineteen actors.
Figure 8-24: The initializing and managing the cast of the Rookery
DEVICE CONTROL AND MODELLING THE USER

Up to this point in the specification of the Rookery, little attention has been paid to the mechanics of how the user will witness and interact with the virtual world on a physical level or how the individual pieces of setting, cast and user can be harmonized into a finished application. This is the next issue to address.

Two versions of the Rookery application were created. The first was a desktop version which relies on the keyboard and mouse for user control. The other was designed to support a head-mounted display and Polhemus tracking sensors. The modular nature of the SHADOW paradigm allowed these two variations to share all but three modules and even in one of these cases it was simply a matter of replacing one subsystem with another where both subsystems had the same logical interface just different internal processing.
System: DeskNav
Input Events: SHADOW_FrameComplete
Output Events: (Not applicable to systems)

![Diagram of System: DeskNav]

**Figure 8-25:** The main system definition for the desktop version of the Rookery

Figure 8-25 shows the main system specification for the desktop version of the Rookery application. A quick inspection of its design reveals that only purpose behind this specification is to integrate and activate the underlying subsystems that constitute the Rookery application itself. The augmented transition network of the specification is used to stagger the activation times of each module so that race conditions with respect to memory allocation and utilization can be avoided. As each module stabilizes it raises an event to inform the system it is safe to activate the next segment of the application.
The first subsystem to be initialized in the application is the desktop input/output controller module. This subsystem is responsible for managing a graphic output channel as well as monitoring external events and translating these events into meaningful signals within the virtual world.

To perform these tasks, the subsystem employs several subsystems of its own. The first, VRPort, is a SHADOW Talk module dedicated to creating, initializing and maintaining a communication channel with the Performer graphics system as well as the local X windows system itself. The second system, MouseHand, is used to interpret raw mouse position and button status information and to derive an equivalent hand position and activity in three dimensional space with respect to the user’s viewpoint. The final contained subsystem is the controller for the user’s hand itself and is responsible for calculating the hand’s velocity vector as well as for determining what, if any, object in the world is being affected when the user issues a grab or release request.

In addition to its contained subsystems, DesktopIO utilizes a demand output link, AdjustView, to monitor for keyboard input and react accordingly. While, as the name implies, the primary goal of the link is to provide the user with a means of adjusting his or her viewpoint within the virtual world, the link also scans for requests that result in altering the systemic behavior of the penguins such as the parameter Wild, which governs how each of the penguins reacts to being picked up by the user.
Subsystem: DesktopIO

Input Events: (none)

Output Events: ChannelReady, ShutdownRequested

Figure 8-26: The desktop I/O controller module

Like the X Window interface utility developed for the widget set example and reused for each of the applications in Section VII, VRPort was designed to provide generic functionality to any application seeking to use the Performer system as a 3-D rendering engine. By embedding
the module in an application and activating the link, the subsystem creates a channel to the Performer engine, opens a window, initializes a scene graph that other subsystem may append to, and begins a regular cycle of polling for discrete input events and transmitting frame refresh requests. The VRPort subsystem is used by both the desktop and head-mounted versions of the Rookery. Its SHADOW Talk specification is given in Figure 8-27.
Subsystem: VRPort
Input Events: (none)
Output Events: ScreenReady, MoveUp, MouseDown, KeyPressed

This system opens a Performer window and maps it to an X window to allow X input events (such as button pressed and keyboard input) to be serviced. Additionally, once channels are initialized, this subsystem deals with servicing the X event queue synchronizing the frame managers until directed to shut down.

Figure 8-27: The VRPort device Performer driver interface
Subsystem: MouseHand
Input Events: (none)
Output Events: GrabActive, GrabOver

Figure 8-28: The MouseHand controller module

The MouseHand controller module, shown in Figure 8-28, was designed to allow a two dimensional pointing device with buttons act as the control unit for a three dimensional manipulator within the virtual world. This was done by assuming that the user's hand within the world commanded a range of motion that could be mapped to the surface of a manipulation sphere relative to the user's viewpoint. The X and Y positions of the mouse were used to control the spherical coordinates of the hands location, while mouse buttons were used to control the sphere's radius. An additional mouse button was used to allow the user to issue grab and release requests which were serviced by the Hand subsystem.
The Hand subsystem is common to both the desktop and head-mounted variations on the application and is used to keep track of the dynamic motion of the user's hand within the world as well as to service grab and release requests in behalf of the user. Figure 8-29 shows the design of this module.

**Subsystem: Hand**

*Input Events: GrabActive, GrabOver*

*Output Events: Gotcha, Released*

---

Figure 8-29: The Hand subsystem
The Hand subsystem consists of: two demand processing links, AddHand and Drop; a dynamic, polled processing link, Grab; a dynamic output link, PositionHand; and, an input link, GetVelocity. AddHand is used to create the graphical object that will represent the user's hand within the Performer scene graph. Drop is used to signal a held object when it has been released from the user's grip. Grab is used to query the underlying model to find objects in close proximity to the user's hand when a grab request is encountered and, if found to notify the object that it has been seized. PositionHand is used to synchronize the position of the hand's avatar within Performer's scene graph with the actual position of the hand within the model. Lastly, GetVelocity is used to monitor changes in the hand's position over time to calculate a velocity vector which is used by other subsystems when attempting to toss objects about.

The augmented transition graph of Figure 8-29 is structured to limit the number of objects the hand can carry. When a grab request comes in, the module goes into a seek mode trying to find a nearby object to grab. If one is found, the subsystem enters a busy state and cannot attempt to grab any other objects until the one currently being held is released.

To complete the discussion of the Rookery application and to illustrate the ease with which two variations based on different I/O devices and technologies were developed, Figure 8-30 presents the main system definition for the head mounted variation of the program. This module is structurally identical to that of the desktop version given in Figure 8-25 and only differs in its use of the subsystem HeadMountIO instead of DesktopIO as the coupling for the link Display.
System: HeadNav
Input Events: SHADOW_FrameComplete
Output Events: (Not applicable to systems)

Figure 8-30: The main system definition for the head-mounted version of the Rookery

The definition of this alternate subsystem is shown in Figure 8-31. It, too, shares a common purpose and structural similarity with its desktop counterpart. The only difference lies in its use of a subsystem, IScan, to control the positions of the user’s hand and viewpoint rather than the mouse and keyboard.
Subsystem: HeadMountIO
Input Events: (none)
Output Events: ChannelReady, ShutdownRequested

Figure 8-31: The HeadMountIO module
The IScan module\textsuperscript{36} is another low-level external device interface designed for reading and coordinating spatial tracking information. For completion, its SHADOW Talk specification is given in Figure 8-32.

\textbf{Subsystem: IScan}

\textbf{Input Events: (none)}

\textbf{Output Events: IScanReady}

\textbf{Figure 8-32:} The IScan device interface module

\textsuperscript{36}The name for this module stems from the actual hardware configuration of the platform on which this system was developed and tested. The hosts in question was outfitted with an ISCAN eye-tracking system which also provided Polhemus spacial tracking data. As a matter of convenience, the device interface for the ISCAN system was used by the Rookery application to get Polhemus data.
CLOSING COMMENTS

The Rookery is a complete virtual world with a non-trivial level of complexity. The code examples presented here represent the entire SHADOW Talk specifications for both the desktop and head-mounted display variations on the application including the utility modules needed to interface to external devices and processes such as the Polhemus sensors and Performer rendering engine. More to the point, each variation was specified and implemented using only twenty-four SHADOW modules each and a very modest amount of external C++ code37.

From a temporal standpoint, the Rookery was designed, specified, implemented and tested in a total time span of approximately eighty man-hours. Surprisingly, the largest single expenditure of time was not the SHADOW Talk specification of the world, but the creation of the graphical elements to be used with the Performer rendering engine. The graphic design and implementation of the penguin actor alone (done without the aid of three dimensional drafting tools) consumed over thirty man-hours. The task of creating the SHADOW Talk specifications to layout and control the work took somewhere between twenty-five and thirty man-hours. To put this number in perspective, it has taken forty man-hours to produce the section of this report that describes the finished design.

The goal of this exercise was to demonstrate that the SHADOW System's claims, explored and supported by small test cases in Section VII, continued to be valid when when faced with a

37Most of this code was dedicated to defining custom graphical objects for use with the Performer graphics system, nearly all of the run time processing used in the operation of the application came directly from the code of the SHADOW link bodies or arose as a direct consequence of the SHADOW run time engine. A more detailed discussion of the language/functionality breakdown can be found in Section IX: Results.
large scale, non-trivial, non-WIMP application. This goal has been achieved. Moreover, this exercise has served to exemplify the expressive power and comparative simplicity of non-WIMP interface development under the the SHADOW paradigm.
SECTION IX: RESULTS

OVERVIEW

The SHADOW System was designed to be a specification language suitable for defining and implementing non-WIMP style user interactions. Its architecture was to have included features and techniques which explicitly address the types of issues and concerns typically encountered when developing applications such as virtual worlds, gesture recognizers, other non-traditional user interfaces. Many of these issues were defined and refined in Section III. The SHADOW paradigm, its language features, and design methodologies were presented in Sections III, V, and VI. Test applications, designed and implemented using those features, were discussed in detail in Sections VII and VIII.

This section will review and distill the information previously presented to correlate how the SHADOW System satisfies the needs detailed in Section III and how the system's claims of support were verified.

ADDRESSING CONCEPTUAL CONTINUITY

Processes in the real world may be seen as collections of both discrete events and continuous relationships. The fact that these processes are eventually to be simulated on a digital computer
(and therefore totally discretized) should not be allowed to interfere with the designer's intuitive understanding of the way processes and events interrelate. It is this conceptual understanding that should be captured in the design process, not the mechanics of how a programmer might implement a digital approximation of it.

In light of this, SHADOW Talk supports specification mechanisms that allow interaction object behavior and manipulation to be described in both continuous and discrete terms. It provides sufficient flexibility so as to allow the designer to draw from both domains concurrently and create hybrid specifications which closely align to his or her conceptual model of the underlying behavior being described. The task of translating these models into executable code is the responsibility of SHADOW System itself, further detaching the designer from the actual, platform-specific implementation details.

The SHADOW paradigm of event controlled, continuous constraint management is evidenced in nearly every example of SHADOW Talk code. The connectivity lines and links of the dynamic data flow graphs are graphic indicators of the continuous relationships, dependencies and data transformations of a module's governing behavior. The augmented transition graphs rigidly define which transformations and relationships are active at any given point in time. State transitions within these graphs clearly define how each module is to react in the face of discrete event phenomena.

To put this in more concrete terms, consider the specification that allows the user to pick up and toss a ball in the Rookery application. The ball is a passive object, it has no motivation of its own and, as such, it contained no controlling logic for how it should fly through the air.
The world, however, is constrained by gravity which affects all objects at all times. To throw a ball in this world, the user must first grab it (discrete action), hold onto it while moving his or her hand in space to build forward momentum (continuous action), and sharply release it (another discrete action). Once free to fall, the continuous constraint of gravity pulls the ball downward in a parabolic arc until the ball is once again at rest atop a supporting object. This scenario describes both the conceptual model of how a ball is thrown from a physical standpoint, as well as the exact logic of how this behavior was simulated in the Rookery.

Similarly, many processes in the real world conduct themselves in parallel with one another. Some such processes interrelate, possibly precipitating new processes, while others proceed essentially independently of one another.

When modelling behaviors which are innately parallel and independent of other processes within an application, the SHADOW System both allows and encourages the designer to isolate the individual process specifications with the understanding that the behaviors they represent will eventually be demonstrated concurrently. The issue of concurrent execution within a uniprocessing environment is the responsibility of the UIMS working in concert with the operating system and entirely detached from the user interface design process.

This design philosophy (and its associated run time engine support) lay at the core of the design of the Rookery example. The individual actor subsystems were designed with isolated animations and agenda under the assumption that they would go about fulfilling their respective roles in parallel with one another. Where concurrent processes interrelated, such as in the ensemble troupes or the alternating steps of the penguins' gait, the synchronization and data
sharing relationships were explicitly declared as state transitions and/or data flows. Application and enforcement of these relationships was left to the SHADOW run time engine to resolve.

**ADDRESSING AND EVALUATING REAL TIME BEHAVIOR**

Non-WIMP systems, such as immersive virtual reality, are often faced with the challenge of producing the best approximation of the desired output within the physical time constraints of a fixed frame rate on a given hardware platform. The SHADOW System is cognizant of these trade-offs and provides an infrastructure through which the interface designer may specify both the ideal behaviour of the system as well as contingency plans and guidelines for process decimation should a given hardware platform prove incapable of maintaining an acceptable throughput rate. Within the SHADOW paradigm, the designer is removed from the implementation details for a given platform and afforded a level of abstraction wherein the designer's responsibility is to provide the UIMS with an assortment of alternative algorithms of varying time complexity along with hints and preferences as to the optimal situations under which each alternative should be explored. The frame by frame decision as to which algorithms to apply and the ability of the overall system to meet its output deadlines is the responsibility of the UIMS itself.

This is accomplished by use of the run time level monitor facilities within the SHADOW run time engine itself. The engine routinely monitors its own performance and updates an internal flag which indicates the system's current processing load and its ability to meet its deadlines. This flag is available for use when specifying the condition under which any given link should
be marked as active. Under this scheme, the designer does not need to be concerned with the frame to frame performance of the actual system on any given hardware platform, he or she need only specify the desired output cycle rate and a variety of contingency plans for the system to follow if the rate is not being met.

This ability was explicitly tested with a somewhat contrived SHADOW application loosely modeled in the structure previously given in Figure 6-8. The application was tasked with performing an operation on a random data set using three possible algorithms with different running times. At run levels (0-2) the system used the 'ideal' solution with a run time of $n^2$. At run levels (3-5) the system was specified to use an approximation technique with running time $n \cdot \log(n)$. As a last resort, at run level (6-9) the system was given the option of using a rough approximation algorithm which ran in linear time. In addition, once the system had reached run level 8, it was directed to further reduce processing load (and accuracy) by reducing the size of the data set to one quarter of its original size.

In parallel with this processing, a load link was added to simulate the system becoming bogged down in other tasks within the application. The demands of this link were structured to place a linearly increasing load on the CPU for approximately nine seconds followed by a linearly decreasing load for another nine seconds before deactivating itself.

This application was then told to maintain an output cycle between 20 and 50 frames per second. This was achieved by setting trip thresholds for the run time engine of 24 and 48

---

38By current standards, this is both a broad range and a fast frame rate, 10 to 15 fps is far more common. The bounds were exaggerated to present results that better illustrates the conceptual behavior of the system rather than its mechanics in practical operation. Even so, these experiments had to be conducted on a Macintosh Quadra 650 (Motorola 68040 CPU) because the processing loads embodied by this application were too small to be of significance in the face of modern computing power.
cycles per second\textsuperscript{39}. Whenever the output rate dropped below the lower bounds, the run level flag was to increase to indicate that timing was becoming an issue and, where available, less computationally intensive algorithms should be substituted in the overall dynamic data flow graph. Should the rate rise as high as the upper bounds, the run level flag is decremented, indicating that the system has free time on its hands and that additional processing can be performed to improve the quality of the system's outputs.

For the purposes of this experiment, the run time engine was modified to report its current run time level and projected frame rate once per output cycle. This information was then compiled in to performance charts like the one shown in Figure 9-1.

![Figure 9-1: The run time behavior of the decimation test program](image)

The heavy dark curve of Figure 9-1 represents the current projected output cycle rate with respect to time. The shaded areas on the bottom of the graph indicate the active run level of the

\textsuperscript{39}The choice of these thresholds is somewhat arbitrary and can be set at run time when the process is first being initialized. In general the trip levels should define a slightly narrower range than the desired output rate to give the system time to react to processing load changes before a deadline is actually missed.
system at the time the data was collected. In this sample execution, the run level can be seen to increase each time the frame rate drops below the lower threshold. In addition, the effect of these changes on the performance of the system is evident by the vertical jumps in the curve when the system performs algorithmic substitutions in response to the new run levels. For example, three seconds into the execution, the cycle rate had dropped to 20 and the run level had risen to 3. This resulted in the automatic substitution of the \( n \lg n \) algorithm in place of the more expensive polynomial time one. The reduction in processing, in turn, resulted in an increased cycle rate.

![Graph](image)

**Figure 9-2:** Overall performance resulting from automated decimation

To minimize the effects of random variances in the data set and pre-emption from the operating system itself, a dozen executions of the test program were run, and the resulting data averaged into the performance graph shown in Figure 9-2. The goal of the exercise was to demonstrate that the run time engine could keep the output rate bounded between 20 and 50 hertz (the slightly narrower range of the trip thresholds of 24 and 48 hertz were selected to give the system time to react before the desired bounds were violated). In general, the system was
successful in this endeavor. As the load increased, the system reacted by trimming other processing burdens, and, as the load eased, increased processing time was allocated to those processes which could benefit from it. The only time the system failed to meet its goal of at least twenty cycles per second came at the very midpoint of the test when the external processing demands were at their peak and the run time engine had already exhausted all of the available contingency plans provided in the specification.

Meeting a given hardware platform's frame rate is only one of the issues facing the next generation of interfaces. The concerns of real time management are further complicated by the introduction of full duplex I/O commonly encountered in many forms of non-WIMP interactions. Many such systems do not have the luxury of simply waiting for a particular discrete event and updating their output accordingly and, instead, must actively monitor continuous streams of data trying to identify probabilistic tokens of context-sensitive, semantic value while simultaneously maintaining a continuous stream output.

To address these complexities, the SHADOW System provides both a specification language that allows isolation of user monitoring and data model updating activities with respect to cognitive scope, while simultaneously providing a run time engine which is sensitive to the temporal correlation between these tasks. The system allows the user interface designer to use temporal abstractions to specify time management and synchronization guidelines between otherwise isolated threads, leaving the low level details of cycle utilization to the run time engine.
The run time engine itself is optimized to perform lazy evaluations so that in each output cycle only the links whose outputs are actively needed are ever calculated. Moreover, the internal implementation of the links themselves employ a dynamic programming philosophy to trade memory consumption for reduced processing time.

**ADDRESSING SCALING ISSUES**

As with any software development, as systems grow in size they run the risk of becoming unmaintainable. While there are a great many contributing causes to this phenomenon, one of the simplest is a lack of traceability; The SHADOW System acknowledges this risk and strives to mitigate it by use of a visual description language and strong support for modular development practices.

While diagrams and visualizations have traditionally been regarded as beneficial for conveying large amounts of information quickly and efficiently, visual languages have often been criticized for their inability to scale. Even simple notations, such as state diagrams, tend to sprawl across the page as the number of states increases. Adding transitions to a complex state diagram may create a network which cannot be rendered in planar space without incurring crossed arcs and raising the visual complexity of the diagram.

These factors were a serious concern during the design of the SHADOW System and, as a result, mechanisms have been built into the specification language to reduce the likelihood that
individual diagrams will become overly complex as the system is applied to large scale programming problems. First, the augmented transition graphs used by SHADOW Talk are non-deterministic, affording the designer the ability to reduce the required number of states for some automata. In addition, the SHADOW System supports modular, hierarchical development allowing the cognitive load to be scoped to small, manageable problems at each phase of the specification.

To illustrate the scaling potential for the SHADOW Talk language, consider the trivial "Hello, World!" application presented in Section VII and the vastly more complex virtual world of the Rookery described in Section VIII. The Hello application possessed two states, two links, and one transition. In contrast, the average subsystem in the Rookery application specified two states, four links, and one transition. Across the entire application, the most complicated augmented transition graph encountered, the penguin controller module shown in Figure 8-14, only contained five states and ten transitions. The award for the most cluttered data flow graph specification goes to the Cast initialization subsystem shown in Figure 8-24. This subsystem contained eleven links, but this complexity is mitigated somewhat by the fact that its associated augmented transition graph only possessed two states and a single transition.

Beyond the issues of size and complexity at the unit level, as the magnitude of a system grows, so does its needs for inter-module coupling and communication. In the case of interfaces which are event driven by nature, as the breadth of the interface increases, so does the number of event producers and consumers. This raises questions with respect to the programmer's ability to address issues of event propagation and scoping.
To address these concerns, the SHADOW Talk language provides mechanisms to allow the designer to discriminate between events based both on the class of event, as well as the event's point of origin. Additionally, it allows the designer to provide hints and directives in the form of event masks which control the generation and propagation of events throughout the system. This affords the design direct control of the module's behavior without becoming bogged down in the low-lying complexities of event management. These features have been applied and tested in multiple examples, and have been found to succeed in ensuring that desired events are serviced promptly by intended consumers while disinterested parties are kept oblivious to the activity.

ADDRESSING EXTENSIBILITY

The SHADOW System was designed to be both flexible and extensible enough to adapt its functionality with the evolving needs of non-WIMP user interfaces. The areas which have demonstrated the most serious volatility as these interfaces grow are the handling of input devices and output channels and the introduction of new interaction objects.

The SHADOW System is sensitive to this concern, and provides mechanisms for dealing with the introduction of new devices in a manner consistent with the overall structure of the UIDL. This is achieved through the use of input and output links encapsulated within reusable device driver subsystem modules. This allows external devices to be defined and integrated into an interface specification smoothly, consistently and in a reusable fashion without requiring any modification of the underlying UIMS.
Several examples of this extension style were evidenced in the sample applications presented in Sections VII and VIII. These examples included: an interface to X Windows; an interface to the Performer graphics engine; a device driver for Polhemus spacial trackers; and, a hybrid device driver combining a spacial tracker with X Window mouse buttons to form a poor man’s data glove. In fact, the SHADOW Talk language has no predefined devices of its own to outgrow, all input and output throughout the UIMS is conducted through links to external devices and or drivers.

Additionally, the existence of new input and output devices gives rise to the definition of novel interaction objects. The SHADOW system recognizes this and provides facilities for the definition, implementation and reuse of highly customized interaction objects. This allows the application scope of the UIMS to be readily extended without compromising backwards compatibility. A demonstration of its ability to fulfill this claim was presented in part in Section VII, wherein a library of two dimensional widgets was created and later employed to create a text editor application.

**SUPPORTING MODULAR DESIGN**

The need for scalability and extensibility required the SHADOW System to come to terms with many of the same issues faced by any high level programming language. Foremost of these issues was the need to provide facilities which support functional decomposition of user interface elements, decoupling of abstract processing threads and the application of modular design principles.
To support these goals, the SHADOW Talk language includes provisions for data hiding, encapsulation, data abstraction and event abstraction. At the highest layers of abstraction, these properties are achieved through the use of system, subsystem, and generic constraint specification units. Each unit has a rigidly defined interface that specifies exactly what information, continuous or discrete, is allowed to cross the module's boundaries. The paradigm supports a containment hierarchy wherein subsystems may exchange information through their interfaces, but the actual mechanics of processing, as well as the true extents of the unit's data requirements, are hidden behind the boundary.

Within the specification of a module, a second layer of data hiding and encapsulation is created with the introduction of the link element. Like the subsystem itself, it has a rigid data interface but may possess its own local storage and processing information.

Extending the concept of data abstraction, the SHADOW Talk language also supports event abstraction. This affords the designer to group, correlate and/or sequence primitive, generic event tokens by employing a succession of handlers to produce higher level, semantically meaningful events optimized for the processing at hand. This was the technique employed in the design of the widgets of Section VII.

**ADDRESSING CODE REUSE**

Many issues relating the the broad area of code reuse are less functions of the language being employed than they are of the documentation, coding standards and level of organization
present in the development of the organization itself. This does not, however, exempt a development language from having to consider reuse practices in its design.

While no language specification alone can guarantee success with respect to code reuse, the SHADOW Talk language strives to provide facilities which will allow code reuse efforts to be successful. It was designed to be accommodating to such efforts incorporating such features as: allowing embedded documentation within specifications; modular isolation; rigid decoupling; and the introduction of customized lexical artifacts which permit the developers to define and impose rigid coding and naming conventions.

At a higher level, code reuse efforts are often hindered not by a lack of reuseable code, but by an inability to recognize that the sought after functionality already exists within a given library [115]. The visual structure of the SHADOW Talk language may also, indirectly, assist in addressing this problem. By delineating the structure (as a data flow graph) from the overall operation (augmented transition diagrams) and low-level implementation details (link bodies), the system has provided a level of meta-information that may assist software analysis tools in classifying and organizing reuse candidates and eliminating redundant library members.

**PRESERVING INTELLECTUAL INVESTMENTS**

Developing a tool whose design and operation is so alien that it takes longer to figure out how to apply the tool to any given problem than it would to solve the problem manually serves no one. In an effort to avoid this scenario, the SHADOW System sought to capitalize on existing
notations, concepts and standards whenever the opportunity presented itself. The most obvious example of this can be found in the visual notation for the SHADOW Talk language and in the specifications of local link body definitions.

The SHADOW Talk language was designed to draw upon the established concepts of a state diagram and a digital circuit wiring diagram. While the full notation is distinctly different from either of these predecessors, anecdotal feedback from programmers attempting to learn the language has suggested that couching new concepts within familiar structures has served to provide a stable launching point for exploring and experimenting with the new paradigm.

In addition to exploiting existing visual notations, the design of the SHADOW System sought to preserve and capitalize upon the investments that industry and academia have made in the C and C++ programming languages. This is exemplified by the choice to use a subset of the C++ syntax model as the base specification language for defining the internal, procedural processing bodies of locally defined links, and by providing a virtually seamless integration of external C++ and C libraries. These decisions proved to be invaluable for the expert and novice users alike, especially when faced with tasks such as integrating SHADOW functionality with existing applications and/or system services.

ASSESSMENT OF THE EXPRESSIVE POWER OF SHADOW TALK

Implicit in all of the foregoing issues is the idea that an appropriate language for a particular programming task is one that has the expressive power to allow the task to be accomplished in
a succinct, maintainable and repeatable fashion without compromising the ability to fine tune necessary low-level details. The claim that SHADOW Talk constitutes just such a language with respect the task of creating non-WIMP interfaces was tested repeatedly by an assortment of sample development efforts, culminating in the creation of the Rookery virtual world example. The successful creation and execution of these programs, however, does not, by and of itself, prove this claim. Experience in legacy system maintenance and migration has repeatedly demonstrated that there is a wide gap between functionally correct code and quality code [92].

To clarify the question of whether the SHADOW Talk language represents an appropriate language for such tasks versus being merely capable of producing the desired results, one must analyze the value added by use of the UIMS. For example, the Rookery could have been written as a complete C++ application and simply placed inside a SHADOW Talk wrapper. Such an implementation would have demonstrated that, yea verily, the language can be used to create a virtual world but would offer no value added\textsuperscript{40}.

The Rookery application, as presented in Section VIII will be used to illustrate the benefits of a proper SHADOW Talk development effort. The specification for the entire application, including device drivers, autonomous and articulated character management, continuous and concurrent animation, and a rudimentary physics engine, was accomplished using only two dozen SHADOW Talk modules and a small collection of external support libraries written in C++. The exact decompositions of the subsystem was discussed in detail in Section VIII. A detailed breakdown of the C++ elements of the application is provided in Figure 9-3.

\textsuperscript{40}In practice, this technique would actually have an overall negative effect on the application in that the program would be incurring all of the overhead of the SHADOW System without exploiting any of its benefits.
Support Code:

- Performer graphic object definitions: 2,645
- Device driver interface code: 283
- Underlying application code: 411
- Total external C++ code: 3,289

Embedded Code:

- Code associated with utility subsystems: 165
- Code associated with Rookery-specific modules: 606
- Total embedded C++ code: 771

Total lines of C++ used in the creation of the Rookery: 4,060

Figure 9-3: C++ utilization in the design of the Rookery

In the final design and implementation of the Rookery application, the programmer was only responsible for creating approximately four thousand lines of C++ code; the rest of the system's functionality came from the twenty-four SHADOW Talk specifications and the base operation of the run time engine. Even the modest four thousand line count is somewhat inflated in that it includes every piece of source code compiled into the finished product, and does not take into account the fact that in a production environment, many of these elements would come from standard libraries of modules, and would not be the programmer's direct responsibility to create or maintain. Likewise, this figure includes a costly charge for the tedious lines of array initialization incurred in the graphic design of the penguin, weight, and balance objects. If these items were to be ignored, the total lines of C++ employed to create and operate the Rookery example drops to 1,017 lines.

---

41 This value includes 2,121 lines of code used to define and initialize the point, facet, and color arrays that make up the custom shapes which comprise the various parts of the penguin actor's anatomy.
42 This includes subsystems like VRPort, which is used to open a channel to the Performer engine and to read mouse and keyboard inputs. These subsystems are included here for the sake of completeness but represent modules a developer would typically include from a standard library rather than create from scratch.
The fact that two dozen diagrams and a thousand lines of C++ can be used to create an entire virtual world with the features and complexities of the Rookery strongly suggests that the SHADOW paradigm offers distinct benefits with respect to creating a succinct system definition. To complete the argument in favor of the SHADOW Talk language, the overall maintainability of this system definition should be considered. While a thousand lines of code is not a particularly large effort by current standards, code fragments as small as a hundred lines can become maintenance nightmares if they are convoluted and poorly written. Likewise, simply stating how many SHADOW Talk diagrams are needed in a specification does little to convey any sense of the level of complexity associated with each schematic.

The question of what constitutes maintainability for a visual language (or any language for that matter) is beyond the scope of this work. However, certain statistical information taken from the Rookery example may be used as the basis for a brief discussion on managing the programmer’s cognitive load. This data is presented in Figure 9-4

<table>
<thead>
<tr>
<th>C++ Code Metrics:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Lines/Procedure (external)</td>
</tr>
<tr>
<td>Average Lines/Link Body</td>
</tr>
<tr>
<td>Average Cyclomatic Complexity (external):</td>
</tr>
<tr>
<td>Average Cyclomatic Complexity (internal):</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subsystem Measures:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average States/Subsystem:</td>
</tr>
<tr>
<td>Average Transitions/Subsystem:</td>
</tr>
<tr>
<td>Average Links/Subsystem:</td>
</tr>
</tbody>
</table>

**Figure 9-4:** Complexity measure for the Rookery.
To maintain a system, a programmer must first be able to understand its design and operation. This understanding is a function of the programmer's experience and expertise, the coding standards under which the system was developed, the level of documentation available, and the size of the cognitive load that the programmer must internalize at any point when reviewing the implementation. This last point, being the one most closely related to the code itself, serves as the crux of the loose argument that if the cognitive load can be scoped to reasonable extents, then specifications written in SHADOW Talk have the potential to be maintainable.

According to Figure 9-4, the average C++ library function used by the Rookery has about twenty-five lines in it. Similarly, the average local link definition consists of about twelve. While exact thresholds for what constitutes an appropriate cognitive load factor is an issue open to debate, many investigators [8][92][93] tend to agree that more than one hundred lines is too big and less than fifty is good. The C++ code of the Rookery falls well into the acceptable range.

In a similar vein, the cyclomatic complexity measure is also open for debate by industry and academia as the magic number that differentiates good code from bad. This debate, however, tends to range over the area from ten to twenty as the general cut off for being too complex. Values in the range from one to four are generally regarded as ideal.

With respect to visual complexity, graphical notations are notorious for becoming large and unwieldy [61][71][104]. As anecdotal evidence that this is not required to be the case, Figure 9-4 reports that for the whole of the Rookery application the average module only employed four links, two states and two transitions. Anyone familiar with automata theory can attest that
a state diagram consisting of only two states and two transitions rarely represents a significant mental challenge.

The example of the Rookery demonstrated that the SHADOW Talk language was capable of creating large scale applications employing non-WIMP interactions. Moreover, it demonstrated that use of the SHADOW paradigm resulted in a specification which was relatively quick and easy to design and translated directly into executable code. The specification modules themselves, while small and succinct, retained the ability to perform low-level manipulations where needed, and combined to produce higher levels of abstraction without incurring higher levels of complexity. In light of the arguments and evidence presented in this section, it is reasonable to declare the SHADOW paradigm a success.
SECTION X: CONCLUSIONS

The SHADOW System was intended to provide a specification paradigm and tool infrastructure to facilitate the definition and implementation of applications featuring non-WIMP style interactions. It was designed to support programming in the large without sacrificing low level control of the specification effort, and was to have introduced an abstraction mechanism to allow a level of conceptual continuity between real world intuition and simulation specification. Its run time engine was to be cognizant of the unique classes of performance and other problems faced by non-WIMP user interfaces and was to have provided facilities to address these needs transparently, without burdening the application designer with the details of their internal operation.

The system and its paradigm were tested against each of these claims and found to be successful and robust. The primary flaw found in the initial proposal offered in Sections IV, V, and VI was that the system was over-designed. An assortment of small features, whose inclusion seemed to make perfect sense based on a C++ programmer's mind set[43] and common practice in the field, turned out to be either unneeded or undesirable in the final application of the tools and techniques. For example, semantic variables (a concept borrowed from previous work on the PMIW System by Jacob) and monitored properties were incorporated into the design in that they seemed to serve an important purpose and prevailing wisdom (that being a

---

[43] Initial experiments with teaching students to accept and apply the SHADOW paradigm property has suggested that mastering the language is less a question of syntax than it is one of perspective. There is a saying "To speak French, one must think French"; this axiom is equally true of the SHADOW Talk language. Just as Pascal programmers first learning C had a tendency to translate familiar code on a token for token basis without restructuring into the new language or exploring the more unique opportunities C offered, students learning SHADOW must first learn that the SHADOW System constitutes a complete development paradigm and not simply a visual wrapper for traditional coding techniques.
wisdom forged by years of experience with semaphores and callbacks) saw them as an obvious and necessary part of the solution. In practice, outside of code specifically contrived to test their functionality, no practical purpose for either of these features has been found. The tasks they were originally intended to address are, in practice, performed by other mechanisms which provide cleaner, more powerful and more traceable interfaces to external applications and processes.

Extraneous features aside, the SHADOW System has been quite successful in achieving its goals and meeting the needs of non-WIMP user interface designers. Moreover, individual experiments and applications of the system have suggested that expressive power of SHADOW Talk and its associated design paradigm may be applicable to more than just the field of human-computer interaction. Programming problems similar to those successfully tackled by SHADOW applications can be found in fields like computer animation, signal and image processing, parsing, machine learning, neural networks, robotics, and other arenas where parallel process specification, task rendezvous, real-time constraint solving, non-deterministic processing and/or cyclic feedback mechanisms represent desirable functionality. While it is unlikely that the SHADOW Talk language is in a position to revolutionize any of these fields, a study into the costs and benefits of its application to these dissimilar domains warrants future investigation.
APPENDIX A: THE SHADOW SCRIPT SYNTAX

The SHADOW Talk is a graphical specification language designed to allow complex program and data flow definitions to be generated by directly manipulating iconic design elements. While affording the programmer a convenient forum for conveying large amounts of information, the diagrammatic specifications produced are of little use to the traditional, line-based UIDL compiler component of the SHADOW UIMS. The SHADOW Script language was designed to bridge this gap.

SHADOW Script is a text based mark-up language used both to render and represent SHADOW Talk specifications. It is the intention of the UIMS that productions within this language be generated automatically by a visual editor and be consumed internally by the UIDL compiler. The motivation behind using a human readable (and writable) format was to facilitate ease of debugging during development of the UIMS itself and to allow the compiler and run time engine to be tested independently of the state of the visual editor development effort.

The notation used in this appendix is a variant on BNF. Each production form is given in an enlarged font and followed by a colon. The expansion of the non-terminal is then given on the line immediately below and indented. If multiple expansions exist, each is presented on a separate line. Optional elements of a production are enclosed in braces. Explicit terminal values are presented in monospaced font and boldfaced. Terminal, non-keyword tokens generated via lexical analysis are presented in monospaced font and italicized. All productions should be in left-recursive form.
The productions below are listed alphabetically. The initial production form for all modules is called Shadow-Module.

---

Activation-Class:

DEMAND

DYNAMIC

This tag is used as part of a link definition to specify the activation class of the link. Dynamic links are re-evaluated each time their activation flag is set and their output values are needed within the SHADOW System. Demand links perform a discrete operation and are fired only once each time their activation flag is set.

---

Binding-Scope:

Target-Object-Spec { Binding-Scope }

Filter-Function { Binding-Scope }

This tag is used as part of a generic constraint specification to define the scope of its activation and association to target objects within the application.

---

Boolean-Flag:

TRUE

FALSE

This tag is used to explicitly restrict the range of answers to bipolar questions and flag values.

---

Appendix A
Constraint-Internals:

Binding-Scope Subelement-List

This tag is used as part of a generic constraint specification to define both the scope and internal functionality of the constraint module.

Constraint-Reference:

<constraint> Filename

This tag is used as part of a system or subsystem definition to specify the inclusion of a generic constraint at the module's level within the containment hierarchy. The actual programmatic definition of the constraint in question should be found in the file named Filename.

Data-Type:

Identifier(Token)

This tag is used to specify the C++ data type of a variable within a variable specification. Currently, the SHADOW System does not explicitly address the use of programmer specified abstract data types within the plugboard specification. However, the generic lexical token identifier has been used here in place of a more specific expansion so as not to preclude such enhancement in the future. An Identifier is defined to be any contiguous string of letters, digits, dollar signs, and underscores which begins with a non-digit.
Destination:

Element-Name

This tag is used as part of a state transition definition to specify the name of the destination state for the transition. The element named in this case should be lexically identical to an existing state name within the module's specification.

Element-Name:

Identifier(Token)

The element name token is a generic production for any named component of a subsystem and is translated into an element of one of three symbol tables (depending on context) during code translation. The lexical parser defines an Identifier to be any contiguous string of letters, digits, dollar signs, and underscores which begins with a non-digit.

Enable-Flag:

ALWAYS

VIRTUAL

Identifier(Token)

This tag is used as part of the link and state definitions to specify the condition under which a given link or set of links will fire. The token ALWAYS indicates an unconditional, continuous relationship between the input and output of a link and is only a valid production when the enable flag is referring to a link definition. The token VIRTUAL indicates that the link in question does not actually exist statically and is being used only as a visual placeholder for zero or more identical links which will be created dynamically at run time. Virtual links will always
have their activation flags changed to ALWAYS or a conditional flag at the moment of instantiation. For links with conditional evaluations (and when defining flags associated with augmented transition network states) a lexical Identifier token is accepted as the name of a flag which must be set to true to make a given link eligible for activation and re-evaluation. An Identifier is defined to be any contiguous string of letters, digits, dollar signs, and underscores which begins with a non-digit.

Event-Name:

DONE

Identifier(Token)

The event name token is a generic production for any explicitly named event which is raised, broadcast, or monitored by a subsystem. The programmer has the flexibility to follow any naming convention desired with respect to uniquely identifying and differentiating discrete actions with two very specific exceptions: the terminal keyword DONE is a special event used internally by a subsystem and is never listed in an interface specification; and, the system generates an event of the form EVENT_property-name when the value of a monitored property changes. The lexical parser does not differentiate between events named by the system versus those named by the programmer and tokenizes both to be simple identifiers. An Identifier is defined to be any contiguous string of letters, digits, dollar signs, and underscores which begins with a non-digit.

Filename:

String(Token)

Appendix A
The grammar is intentionally lax with respect to the exact nature of a file name. This was done to allow for ease of porting between heterogeneous systems. In light of this, the lexical token String has been used in its place. A string is defined to be a stream of ASCII characters delimited by the double quote character.

Filter-Function:

<filter Test-List >

This tag is used as part of a generic constraint binding definition to specify additional restrictions that can exempt subsystems from the effects of the governing constraint.

Flag-Definition:

<flag value= Boolean-Flag > Enable-Flag Flag-Internals </flag>

This tag is used as part of a state definition to specify which activations flags should be set or reset upon entering the current state.

Flag-Internals:

Render-Info

This tag is used to specify internal meta-data regarding the visual placement of flag labels on the augmented transition network diagram and is ignored by the compiler itself.

Flow-Definition:

<def_flow> Flow-Internals </def_flow>

This tag is used to specify the connectivity of elements within a data flow graph.
Flow-Internals:
  Render-Info { Flow-Internals }
  <source> Plug-Node { Flow-Internals }
  <destination> Plug-Node { Flow-Internals }

This tag is used as part of a flow definition to specify both the presentation and function of edges within a SHADOW module's data flow graph. Each alternate expansion should occur exactly once.

G-Box-Spec:
  <box Point Point >

This tag is used to specify a box outline within a graphics specification. The given points specify the upper left and lower right corners of the box within a normalized coordinate system.

G-Line-Spec:
  <line Point Point >

This tag is used to specify a line between the two named end points within a graphics specification.

G-Spline-Spec:
  <spline Point Point Point >

This tag is used to specify a spline between the two named end points and passing through a given (temporal) midpoint within a graphics specification.
G-Text-Spec:

\[
<\text{text } G-Text-Spec-Attributes > G-Text-Spec-Body </text>
\]

This tag is used to specify the placement of a text string within a graphics specification.

G-Text-Spec-Attributes:

\[
\{ \text{orientation= Orient-Type } \} \text{ Point Point}
\]

This tag is used as part of a graphic text definition to specify the orientation and bounding area of the intended text body. The default orientation for all text is horizontal. The two points given represent the upper left and lower right corners of a bounding box within which the text is to be placed.

G-Text-Spec-Body:

\[
<\text{body} > String(Token) </body>
\]

This tag is used as part of a graphic text definition to specify the actual text to be displayed within a given graphical bounds. The string itself is accepted from the lexical analyzer as any collection of ASCII characters.

Graphics-Specification:

G-Line-Spec {Graphics-Specification}
G-Spline-Spec {Graphics-Specification}
G-Box-Spec {Graphics-Specification}
G-Text-Spec {Graphics-Specification}
The graphics specification production is used to specify a collection of vector graphic elements which define the appearance of an icon. A vector system with normalized coordinates was chosen over a traditional bit-mapped approach to facilitate scaling and skewing within the graphical editor.

Icon-Specification:

<icon> Graphics-Specification </icon>

This tag is used as part of the public interface specification to define a graphical icon representation to be used for labeling the subsystem when it is wholly contained within another system. The system also maintains a schematic view of the subsystem to show how its inputs and outputs interconnect with its container but said representation is derived implicitly from the subsystem specification and cannot be customized. If no icon specification is provided, the schematic view will be used in its stead.

Input-Event-Ref:

<in_event> Event-Name

This tag is used as part of the subsystem public interface to specify that a given event has meaning within the subsystem and that if such an event is broadcast to the containing system, it should be propagated down to this subsystem as well.

Interface-Definition:

Appendix A
<interface> Interface-Internals </interface>

The interface definition is used to describe both the presentation and intercommunication between a subsystem and its container. Additionally, it may also specify external ANSI C or C++ libraries to be included at compilation time.

Interface-Internals:

Library-Reference {Interface-Internals}
Input-Event-Ref {Interface-Internals}
Output-Event-Ref {Interface-Internals}
Icon-Specification {Interface-Internals}

This tag is used for declaring the various parts of a subsystem's interface to its containing supersystem and the outside world. NOTE: Syntactically this production allows for multiple icon specifications, in practice, at most one such specification should be allowed.

Library-Reference:

<library> Filename

This tag is used in an interface specification to direct the compiler to include a C or C++ library header file during code generation and compilation.
Link-Attributes:

\[
\text{class= Activation-Class \{ Link-Attributes \} }
\]

\[
\text{function= Link-Functions \{ Link-Attributes \} }
\]

\[
\text{source= Link-Sources \{ Link-Attributes \} }
\]

\[
\text{enableflag= Enable-Flag \{ Link-Attributes \} }
\]

\[
\text{priority= Link-Priority-Code \{ Link-Attributes \} }
\]

\[
\text{maxrunlevel= Run-Level-Code \{ Link-Attributes \} }
\]

\[
\text{minrunlevel= Run-Level-Code \{ Link-Attributes \} }
\]

This tag is used as part of a link definition to list the general attributes that constitute the link's character and behavior within the system. Most of these attributes have default values which are used if no further specification is given. These defaults are: Class = DYNAMIC; Function = PROCESSING; EnableFlag = ALWAYS; Priority = 0; MaxRunLevel = 9; MinRunLevel = 0. In practice, each production alternative should appear, at most, once.

Link-Body:

\[
<\text{body}> \ C++\_Code(AltGrammar) \ </body> 
\]

This tag is used as part of a local link definition to specify explicit programming directives which either enforce constraining relationships between input and output variables, or interface to external entities such as APIs, semaphores, sockets, etc. The C++ code production is accepted by an alternate grammar which inherits a local symbol table containing the link's input and output variables as well as symbols introduced from the C++ libraries included in the

---

Appendix A
SHADOW module's definition. The code accepted here should be in the form of a sequence of statements which may include local memory allocation but not function specifications, type definitions, or compiler directives.

---

Link-Definition:

```xml
<def_link Link-Attributes > Element-Name Link-Internals </def_link>
```

This tag is used as part of a SHADOW module specification to define all forms of links used by the subsystem whether they are defined internally or exist as contained subsystems.

---

Link-Functions:

```
INPUT
OUTPUT
PROCESSING
```

This tag is used as part of a link attribute definition to specify the purpose of a link within the subsystem. Input links are used to monitor external physical devices such as mice, data gloves, cameras, microphones, etc., and as such, are considered to be eligible for re-evaluation by the run time engine even if their declared input variables (if any) have not changed. Output links perform similar interfacing operations and are used by the run time engine as root nodes in generating re-evaluation trees. The keyword PROCESSING is used to identify all other links intended for intermediate data handling.

---

Appendix A
Link-Input-Var:

\[ <\text{in\_var} \ \text{type=Data\_Type} > \ \text{Element\_Name} \]

This tag is used as part of the definition of local links to specify an input variable. An accepted Element-Name becomes part of the local symbol table and becomes available for reference within the body of the specification and may be used as the destination of a data flow specification.

---

Link-Internals:

- Render-Info \{ Link-Internals \}
- Link-Input-Var \{ Link-Internals \}
- Link-Output-Var \{ Link-Internals \}
- Link-Body \{ Link-Internals \}

This tag is used as part of the definition of local links to specify (in full or in part) the presentation and operations of the link. In practice, only one link body should be defined and at most one render information tag should be given.

---

Link-Output-Var:

\[ <\text{out\_var} \ \text{type=Data\_Type} > \ \text{Element\_Name} \]

This tag is used as part of the definition of local links to specify an output variable. An accepted Element-Name becomes part of the local symbol table and becomes available for reference within the body of the specification and may be used as the source of a data flow specification.

---

Appendix A
Link-Priority-Code:

\textit{Digit}(\text{Token})

This tag is used as part of a link attribute definition and indicates the relative importance that a given link be evaluated even when its output is not currently required. The lexical token Digit is defined to be any single ASCII character whose value is between 48 and 57.

Link-Sources:

\texttt{LOCAL}

Filename

This tag is used as part of a link attribute definition to specify where the main body of code defining the link may be found. The token LOCAL indicates that the link-internals production will hold a complete link definition. Otherwise, the link (in this case an entire subsystem being encapsulated as a link) may be found in the source file indicated by Filename. In practice, if a file name has been specified here, the link-internals production should only contain rendering information.

Notes-Body:

\begin{verbatim}
<body> ASCII-Text </body>
\end{verbatim}

This tag is used as part of a local annotation definition to specify the textual content of the annotation.
Notes-Definition:

<def_note> Notes-Internals </def_note>

This tag is used as part of a SHADOW module specification to add an annotation to a SHADOW Talk diagram. This information is of no interest to the compiler and is discarded during code translation.

Notes-Internals:

Render-Info { Notes-Internals }

Notes-Body { Notes-Internals }

This tag is used as part of the definition of local annotations to specify the placement and content of textual annotations on SHADOW Talk diagrams.

Orient-Type:

NORMAL

RISING

FALLING

This tag is used within the graphics text specification (G-Text-Spec) to indicate how a text string should be rendered with its bounding box. NORMAL indicates traditional left to right horizontal orientation encountered in many western natural languages. RISING indicates that the string should be rotated 90 degrees counter-clockwise from normal and read from bottom to top. FALLING indicates that the text should be rotated 90 degrees clockwise and read from top to bottom.
Output-Event-Ref:

\texttt{<out\_event>} Event-Name

This tag is used as part of the subsystem public interface to specify that a given event has the potential to be raised to the containing subsystem and is to be considered a primary channel for sending discrete signals to said system.

Plug-Node:

Element-Name

Element-Name . Element-Name

This tag is used to specify the end point of a data flow specification. The element names specified here should be valid symbols in the local data element symbol table and must conform to the following restrictions: if the former production is used, then Element-Name must refer to some form of locally defined variable; if the latter production is used, the first Element-Name must refer to a locally named link and the second must refer to one of said link's interface variables.

Point:

\texttt{( Float(Token) , Float(Token) )}

This tag is used by the graphical editor to identify a point relative to its bounding area in a normalized space. The float tokens are produced by the lexical analyzer and correspond to the lexical definition of a floating point constant within the C programming language.

\textbf{Appendix A}
Point-List:

Point { Point-List }

This tag is used to specify a non-null set of points. It is used internally to the SHADOW Editor system and is discarded during code translation.

Render-Info:

<render Point-List>

This tag is used internally by the SHADOW Editor application to hold graphic rendering information regarding the layout of visual elements within a SHADOW Talk specification. This information is ignored during code translation.

Run-Level-Code:

Digit(Token)

This tag is used as part of a link attribute definition and indicates a minimum or maximum run level at which a given link should be allowed to be recalculated. The lexical token Digit is defined to be any single ASCII character whose value is between 48 and 57.

Shadow-Constraint:

<def_constraint> Element-Name Constraint-Internals </def_constraint>
This tag is used to specify the nature of the code contained in a physical file and should begin on the first non-blank line of the file. A constraint specification implies that this is a generic binding which will be dynamically scoped to other subsystems based on the existence or absence of certain named properties within the other subsystems' specifications.

Shadow-Module:

Shadow-System

Shadow-Subsystem

Shadow-Constraint

These are the three basic types of modules that may be specified. Syntactically, there is very little difference between a system and a subsystem. Semantically, a system is a subsystem that serves as the application level container for all other subsystems. A constraint contains many of the same elements as a subsystem but also maintains binding and scoping information so that it may be dynamically associated with user interface objects rather than statically bound to explicit scopes as subsystems are.

Shadow-Subsystem:

<def_subsys> Element-Name Subelement-List </def_subsys>

This tag is used to specify the nature of the code contained in a physical file and should begin on the first non-blank line of the file. A subsystem specification implies that this is a generic specification of an element of a larger system and may be instantiated many times at various levels of the main application's containment hierarchy.
Shadow-System:

<def_system> Element-Name Subelement-List </def_system>

This tag is used to specify the nature of the code contained in a physical file and should begin on the first non-blank line of the file. A system specification implies that this is the top level/outermost container in the system and constitutes the starting point for the application.

State-Attributes:

initial= State-Attribute-Settings

This tag is used as part of a state definition to identify the initial state of the system’s augmented transition network. In practice, this tag should be set to false (the default) for all but one state in the network.

State-Attribute-Settings:

ALWAYS

Boolean-Flag

This tag is used to define the valid possible settings for a state’s Initial flag. The setting of ALWAYS indicates that each time the associated SHADOW module is activated, its augmented transition network should be reset to the given state. If set to TRUE, the given state is considered a one-time start state but the augmented transition network is never to be artificially reset. In practice, all but one of the states in the network are to be set to FALSE and exactly one state must be set to TRUE or ALWAYS.
State-Definition:

<def_state State-Attributes > Element-Name State-Internals </def_state>

This tag is used as part of a subelement definition list for a SHADOW module to specify a particular operating state within a subsystem. In practice each SHADOW module must define at least one state.

State-Internals:

    Render-Info { State-Internals }
    Transition-Definition { State-Internals }
    Flag-Definition { State-Internals }

This tag is used as part of a state definition to specify the presentation, connectivity and character of a given state. In practice, there should be at most one render tag associated with each state definition.

Subelement-List:

    Interface-Definition {Subelement-List}
    Notes-Definition {Subelement-List}
    Variable-Definition {Subelement-List}
    Link-Definition {Subelement-List}
    Flow-Definition {Subelement-List}
    State-Definition {Subelement-List}
    Constraint-Inclusion {Subelement-List}

Appendix A
The sub-element list is at the core of all three types of Shadow Module, however, the generic production given here is slightly more generous than the actual syntax should be. Specifically, not all of the available productions according to this list are semantically correct in the context of an actual specification. A constraint module, for example, should not include other constraints within its own definition.

---

Target-Attributes:

\[
\text{<core> Element-Name \{ Target-Attributes \}}
\]

\[
\text{<associate> Element-Name \{ Target-Attributes \}}
\]

This tag is used as part of a generic constraint binding definition to create bindings to existing properties of other subsystems and to assign new properties to said subsystems for use by the governing constraint.

---

Target-Object-Spec:

\[
\text{<target> Element-Name Target-Attributes \text{ </target>}}
\]

This tag is used to define a generic target for a given constraint based on the pre-existence of certain properties. For the purposes of the generic constraint's processing, any selected target object will subsequently be referred to by the token mapped to Element-Name.
Test-List:

\[ \text{test} = \text{C++ Function Call(AltGrammar)} \{ \text{Test-List} \} \]

This tag is used as part of a filter function definition to specify an explicit test function that must return true before a global constraint will be allowed to affect a given target object. In the case of multiple tests being applied, they must all return true for processing to proceed.

Trans-Attributes:

\[ \text{token} = \text{Identifier(Token)} \{ \text{Trans-Attribute-Conditional} \} \]

This tag is used as part of a state transition definition to define the conditions under which a given state transition may occur. The token Identifier refers to the name of a particular discrete event which is being monitored or generated by the subsystem. The conditional production is used to place additional restrictions on how the network should react to the given token.

Trans-Attribute-Conditional:

\[ \text{condition} = \text{C++ Function Call(AltGrammar)} \]

This tag is used as part of the transition attributes tag to define an additional processing call which must return true to allow an associated state transition to occur.
Trans-Internals:

    Render-Info

This tag is used to specify internal meta-data regarding the visual placement of transition arcs and labels on the augmented transition network diagram, and is ignored by the compiler itself.

Transition-Definition:

    <transition Trans-Attributes> Destination Trans-Internals </transition>

This tag is used as part of a state transition definition to describe a potential transition path out of the state currently being defined.

Var-Attributes:

    type= Data-Type {Var-Attributes}
    class= Var-Class {Var-Attributes}
    default= (Token) {Var-Attributes}
    monitored= Boolean-Flag {Var-Attributes}

This tag is used to list the general properties set for any given variable's specification. In practice, each keyword should occur, at most, once. The tag Type is required for all variables. The tag Monitored should only be associated with variables of type PROPERTY. If the tag Class is omitted in a variable definition it will default to the value of EXPLICIT.
Var-Class:

    INPUT
    OUTPUT
    PROPERTY
    EXPLICIT
    SEMANTIC

The variable classification tag is used to specify which of five static categories a particular variable falls into. A sixth category, IMPLICIT, as the name suggests, exists but is not recorded within the Shadow Script language; its existence is inferred by the system during code generation.

Var-Internals:

    Render-Info

The render tag specified here is used to record information used by the graphical editing system for placement and presentation of iconic representations of variables within the data flow graph and is discarded during code generation.

Variable-Definition:

    <def_var Var-Attributes > Element-Name Var-Internals </def_var>

This tag is used as part of a subsystem specification to define all forms of variables used by the subsystem.

Appendix A
APPENDIX B: THE SHADOW LINK DEVELOPMENT LIBRARY

OVERVIEW

Part of the design philosophy behind the SHADOW System was to create a paradigm wherein high level abstractions of interaction objects and behaviors could be created without sacrificing the low level control needed to create interfaces to new I/O channels, widget libraries, and other extensions to the base functionality. Unlike many other user interface management systems, the SHADOW System does not have a predefined pointer and keyboard device channel, a fixed and finite set of event tokens, or a predefined table of available callbacks. Instead, it provides the facilities to allow such things to be built in a customizable and reusable fashion.

These facilities take three forms: the SHADOW Talk language itself which is the primary specification mechanism for all elements within the system; the run time engine which manages the execution of the specifications; and, the event interface and link development library which provide mechanisms for the run time engine to explicitly share information and sequential processing tasks with the SHADOW Talk subsystems. The first two components are described in great detail in the main body of this work. The third and final piece will be described in this appendix.
The event interface used by the run time engine is quite small. Events were intended to allow subsystems to send discrete between one another with the idea being that the subsystems themselves would define what those signals should be. However, the nature of the run time engine's behavior is such that there are certain classes of events that only it can detect but the actual application might wish to be made aware of. For these limited cases, the run time engine is capable of issuing events directly to the application.

Additionally, there are times during the development of SHADOW Talk links and subsystems when the individual elements need to draw upon facilities and data which are beyond the scope of the normal SHADOW Talk declarative specification. This situation is most frequently encountered when defining the bodies of local links using a subset of the C++ language but may also arise when placing conditions on state transitions or filter functions on generic constraints. To address these situations, the SHADOW System provides a collection of C++ compatible functions that may be invoked from the bodies of local links or in other situations where a C++ function call is considered to be an acceptable token by the SHADOW Talk specification.
**UIMS INITIATED EVENTS**

**SHADOW_INITIALIZE** : This event is issued exactly once immediately before the start of the very first output evaluation cycle and is sent to the system level specification module of the application if and only if its input event mask has requested notification. Once transmitted to this level the event propagates down the containment nest as would any ordinary event.

**SHADOW_FRAMECOMPLETE** : This event is issued immediately after the completion of an output evaluation cycle and is sent to the system level specification module of the application if and only if its input event mask has requested notification. Once transmitted to this level the event propagates down the containment nest as would any ordinary event. Access to this signal is particularly useful for implementing double-buffered animations.
RUN TIME ENGINE QUERY FUNCTIONS

SHADOW_Active_System(SubsystemAddress) : This function returns a non-zero integer value if the referenced subsystem is currently marked as active within the main dynamic data flow graph and zero if it or any of its containing supersystems are currently flagged as inactive. This function is particularly useful as a filter function when scoping the application of generic constraints.

SHADOW_Get_Current_Time_Index(void) : This function returns a value of type long which reflects the current value of the run time engine's internal cycle counter. This counter is initially set to zero and is incremented approximately once every 10-15 milliseconds depending upon the clock resolution of the target platform.

SHADOW_Get_Subsystem_Address(void) : This function is used to get the address of the subsystem that contains the link making the call.
VIRTUAL LINK MANAGEMENT FUNCTIONS

SHADOW_Install_Subsystem (SubsystemName, ActivationFlagName) : This is used to instantiate a virtual link within a running system. The two parameters refer to the name of a subsystem previously declared with an activation flag of VIRTUAL and the name of the actual activation flag that is to be associated with the instance of the subsystem. When a new subsystem is created all generic constraints within scope are notified of its existence and any appropriate associate properties are bound to the module. When bookkeeping tasks have been completed, the function returns the address of the new subsystem.

SHADOW_Remove_Subsystem (SubsystemAddress) : This function is used to destroy an instance of a subsystem previously created using the SHADOW_Install_Subsystem facility. When a subsystem is destroyed, all coupling to the dynamic data flow graph are severed and any bindings to generic constraint modules are broken. This function should only be used on virtual subsystems.
EVENT MANAGEMENT FUNCTIONS

SHADOW_Broadcast_Event(EventName) : This call is used to send an event from a link to its associated augmented transition graph within the same subsystem. In addition, if other links within the data flow graph represent contained subsystems which accept the given event, they, too, will receive a copy of the event token.

SHADOW.Raise_Event(EventName) : This function is used to both broadcast an event (c.f. SHADOW_Broadcast_Event() ) as well as to propagate an event upward to containing supersystems. To be eligible to export at each tier the subsystem must include the named event in its output event mask.

SHADOW.Pass_Event(EventName, SubsystemAddress) : This function is used for point to point transmission of events without incurring the overhead of navigating the containment hierarchy's maze of event masks. This utility can be quite useful when designing generic constraints which need to send signals to their target objects but may not be aware of the objects' positions within the overall architecture of the application.
BIBLIOGRAPHY


Bibliography


Bibliography


---

Bibliography
86. Lamure, H., and Michelucci, D., "Solving Geometric Constraints by Homotopy", 
Proc. ACM Solid Modeling '95 Conference, pp. 263-269, Salt Lake City, UT, 1995

87. Larson, J. A., Interactive Software: Tools for Building Interactive User Interfaces, 


Seventh Annual Software Technology Conference for the Warrior, Salt Lake City, UT, April 1995.

Bibliography


---

**Bibliography**


3D Interface Toolkit," Proc. ACM UIST'94 Symposium on User Interface Software 
and Technology, pp. 59-67, Addison-Wesley/ACM Press, Marina del Rey, CA, 
1994.


136. Szekely, P., Luo, P., and Neches, R., "Facilitating the Exploration of Interface Design 
Alternatives: The HUMANOID Model of Interface Design," Proc. ACM CHI'92 

Variables into One-Way Constraint Models," ACM Transactions on Computer-Human 


