Interaction-Based Simulation
by
Gene S. Lee
M.Sc. (Computer Science), The University of British Columbia, 1994
Sc.B. (Computer Science and Engineering), Massachusetts Institute of Technology, 1989
A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
Doctor of Philosophy
in
THE FACULTY OF GRADUATE STUDIES
(Department of Computer Science)
we accept this thesis as conforming
to the required standard
THE UNIVERSITY OF BRITISH COLUMBIA
July 27, 2001
© Gene S. Lee, 2001
In presenting this thesis in partial fulfillment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying for this thesis for scholarly purposes may be granted by the Head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science  
The University of British Columbia  
2366 Main Mall  
Vancouver, British Columbia  
Canada V6T 1Z4

Date July 31, 2001
Abstract

This thesis presents a technique for producing visual simulations — programs that visualize the execution or behavior of time-varying scenes — from software interactions, first-class structures that moderate the flow of information among software components. The building of programs with software interactions gives rise to Interaction-Based Programming (IBP), a programming methodology that separates the concerns of computation from coordination. Software components compute independently, while software interactions coordinate communications. The integration of IBP with the methods of computer simulation produce Interaction-Based Simulation (IBS), an approach to visual simulation that binds the execution of software interactions to the advancement of time. As time advances, software interactions control the flow of information among components and the dissemination of temporal information.

The utility of IBS is presented in two ways: one, through a logical description of the effect of software interactions on software development and systems created using the approach; and two, through empirical evidence demonstrating the ability of IBS to produce a wide variety of programs that encourage the reuse of software.
Contents

Abstract ii

Contents iii

List of Tables ix

List of Figures x

Acknowledgements xv

Dedication xvii

1 Introduction 1

1.1 Difficulties of Creating Visual Simulations 2

1.2 Overview of Existing Approaches 3

1.3 Shortcomings of Existing Approaches 3

1.3.1 Animating Methods 4

1.3.2 Complexity Management 5

1.3.3 Software Reuse 7

1.4 Thesis Statement 8

1.4.1 Interaction-Based Simulation 9

1.4.2 Benefits of Interaction-Based Simulation 14

1.5 Thesis Overview 15

2 Background 17

2.1 Computer Animation 18

2.1.1 Structural Overview 18

2.1.2 Animating Methods 19

2.1.3 Complexity Management 21

2.1.4 Software Reuse 22

2.2 Computer Simulation 23

2.2.1 Structural Overview 24
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3.3 Extended Example Revisited</td>
<td>63</td>
</tr>
<tr>
<td>6 Multi-Modeling Methods</td>
<td>66</td>
</tr>
<tr>
<td>6.1 Forms of Software Interactions</td>
<td>66</td>
</tr>
<tr>
<td>6.1.1 Categories</td>
<td>67</td>
</tr>
<tr>
<td>6.1.2 Time-Varying</td>
<td>68</td>
</tr>
<tr>
<td>6.1.3 Extended Example Revisited</td>
<td>69</td>
</tr>
<tr>
<td>6.2 Scripts</td>
<td>69</td>
</tr>
<tr>
<td>6.2.1 Kinds of Scripts</td>
<td>70</td>
</tr>
<tr>
<td>6.2.2 Declarative</td>
<td>72</td>
</tr>
<tr>
<td>6.2.3 Functional</td>
<td>74</td>
</tr>
<tr>
<td>6.2.4 Extended Example Revisited</td>
<td>76</td>
</tr>
<tr>
<td>6.3 Software Composition</td>
<td>77</td>
</tr>
<tr>
<td>6.3.1 Aggregation</td>
<td>78</td>
</tr>
<tr>
<td>6.3.2 Assembly</td>
<td>79</td>
</tr>
<tr>
<td>6.3.3 Extended Example Revisited</td>
<td>82</td>
</tr>
<tr>
<td>7 General Specification</td>
<td>86</td>
</tr>
<tr>
<td>7.1 Model Graph</td>
<td>87</td>
</tr>
<tr>
<td>7.1.1 Parts</td>
<td>88</td>
</tr>
<tr>
<td>7.1.2 Evaluation</td>
<td>89</td>
</tr>
<tr>
<td>7.1.3 File Format</td>
<td>89</td>
</tr>
<tr>
<td>7.2 Action Graph</td>
<td>90</td>
</tr>
<tr>
<td>7.2.1 Parts</td>
<td>90</td>
</tr>
<tr>
<td>7.2.2 Evaluation</td>
<td>92</td>
</tr>
<tr>
<td>7.2.3 File Format</td>
<td>92</td>
</tr>
<tr>
<td>7.2.4 Extended Example Revisited</td>
<td>95</td>
</tr>
<tr>
<td>7.3 Time Graph</td>
<td>96</td>
</tr>
<tr>
<td>7.3.1 Parts</td>
<td>96</td>
</tr>
<tr>
<td>7.3.2 Evaluation</td>
<td>99</td>
</tr>
<tr>
<td>7.3.3 File Format</td>
<td>100</td>
</tr>
<tr>
<td>7.3.4 Contextual Example Revisited</td>
<td>100</td>
</tr>
<tr>
<td>7.4 Comparison to VRML and JAVA3D</td>
<td>103</td>
</tr>
<tr>
<td>7.4.1 Similarities</td>
<td>103</td>
</tr>
<tr>
<td>7.4.2 Differences</td>
<td>103</td>
</tr>
<tr>
<td>8 Reuse Guidelines</td>
<td>106</td>
</tr>
<tr>
<td>8.1 Interactional</td>
<td>107</td>
</tr>
<tr>
<td>8.1.1 Properties</td>
<td>107</td>
</tr>
<tr>
<td>8.1.2 Contextual Example Revisited</td>
<td>110</td>
</tr>
</tbody>
</table>
8.2 Behavioral ............................................. 112
8.2.1 Properties ............................................. 112
8.2.2 Contextual Example Revisited ...................... 115
8.3 Temporal ................................................. 116
8.3.1 Properties ............................................. 117
8.3.2 Contextual Example Revisited ...................... 120

9 Implementing Animating Methods: A Comparison 122
9.1 Overview .................................................. 122
9.1.1 Implementation and Analysis ...................... 123
9.2 Key-Frame Animation .................................... 124
9.2.1 Cory’s Implementation ............................... 124
9.2.2 Mel’s Implementation ................................. 128
9.2.3 RASP’s Implementation .............................. 129
9.2.4 Comparison ............................................. 131
9.3 Procedural Animation .................................... 134
9.3.1 MAM/VRS’s Implementation ....................... 135
9.3.2 Fran’s Implementation ............................... 138
9.3.3 RASP’s Implementation .............................. 140
9.3.4 Comparison ............................................. 142
9.4 Particle System Animation ............................. 143
9.4.1 ASAS’s Implementation ............................. 144
9.4.2 PARTICLE API’s Implementation ................. 145
9.4.3 RASP’s Implementation ............................. 146
9.4.4 Comparison ............................................. 149
9.5 Summary of Evaluation ................................. 151

10 Implementing Domain Applications: A Comparison 153
10.1 Overview .................................................. 153
10.2 Algorithmic Animation .................................. 154
10.2.1 POLKA’s Implementation ......................... 155
10.2.2 OBLIQ-3D’s Implementation ....................... 158
10.2.3 RASP’s Implementation ............................. 160
10.2.4 Comparison ............................................. 162
10.3 Scientific Visualization ............................... 165
10.3.1 VTK’s Implementation ............................. 166
10.3.2 VRML’s Implementation ........................... 168
10.3.3 RASP’s Implementation ............................. 173
10.3.4 Comparison ............................................. 175
10.4 Summary of Evaluation ................................. 177
11 Easing Application Development: An Assessment

11.1 Managing Complexity with Dynamic Topology

11.1.1 Run-time Configuration

11.1.2 Application to Visual Simulations

11.2 Managing Complexity with Multi-Modeling

11.2.1 Application to Visual Simulation

11.3 Supporting Reuse with the Guidelines

11.3.1 Production

11.3.2 Adaptation

11.3.3 Preliminary Experiences

12 Related Work

12.1 Interconnections in Software Engineering

12.1.1 Semantic Data Modeling

12.1.2 Module Interconnection Languages

12.1.3 Architecture Descriptions

12.1.4 Dynamic Software Architectures

12.1.5 Connectivity Modeling

12.2 Interconnections in Visual Simulation

12.2.1 Computer Graphics

12.2.2 Computer Simulation

12.3 Interconnections in Dataflow Architectures

13 Conclusion

13.1 Visual Simulation

13.2 Summary of the Thesis

13.2.1 Objective

13.2.2 Solution

13.2.3 Contributions

13.3 Impact

13.3.1 Visual Simulation

13.3.2 Software Development

13.4 Future Investigation

13.4.1 Drawbacks

13.4.2 Enhancements

13.4.3 User Studies

Bibliography
## List of Tables

4.1 Two means of passing information from a and b ........................................ 41
4.2 Common query types of first-class operations ........................................... 42
4.3 Attributes of software interactions ......................................................... 44
4.4 Attributes of the software interactions in Figure 4.3 ............................... 46
4.5 Attributes of software interactions ......................................................... 48

6.1 Various kinds of events ................................................................. 67
6.2 Various kinds of scripts ................................................................. 70
6.3 Description of process states .......................................................... 74

7.1 Kinds of model graph nodes ................................................................. 88
7.2 Kinds of time graph nodes ................................................................. 98
7.3 The effects of temporal relations on active and inactive scripts .............. 98

11.1 Various simulations produced with RASP ............................................. 189
11.2 Property violations of the original code for “stampede” and “mesh compress” 191
List of Figures

1.1 Three software components under the influence of two software interactions ........................................ 9
1.2 The three layers of Interaction-Based Simulation .................................................................................. 10
1.3 A software network arises from the arrangement of components with software interactions ................. 10
1.4 The activities for coordination consist of scripts that determine the execution of a software network. ........................................ 11
1.5 As individual scripts execute, they produce topological changes to the inter­
connections amongst software components. .............................................................................................. 11
1.6 An interaction constraint that relates the sum of compA and compB. ................................................. 12
1.7 An example of a multi-model simulation ............................................................................................... 13
1.8 The general specification partitions a visual simulation into three graphs .......................................... 13
2.1 Four phases of simulation development .............................................................................................. 24
2.2 The methods of temporal management [169] ..................................................................................... 25
2.3 An example of a multi-model system ................................................................................................ 28
3.1 A software network of a simple animation .......................................................................................... 32
3.2 The software network of the extended example .................................................................................. 33
3.3 An illustration of the contextual example ........................................................................................... 34
3.4 Four steps of application development with RASP ............................................................................ 35
4.1 A typical software network .................................................................................................................. 37
4.2 A component with five unidirectional ports ....................................................................................... 40
4.3 The general example revisited ............................................................................................................ 46
4.4 The coding of the general example of Figure 3.1 ............................................................................. 51
4.5 A meta-script for animating the movements a robot ............................................................................. 52
4.6 The timeline of Figure 4.5 .................................................................................................................... 52
4.7 An interaction-based program ............................................................................................................. 53
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.8</td>
<td>The three phases of program execution with IBP</td>
<td>53</td>
</tr>
<tr>
<td>4.9</td>
<td>A comparison between a program that changes topology and a program that consists of a single network</td>
<td>55</td>
</tr>
<tr>
<td>5.1</td>
<td>Exemplary definitions of three virtual ports</td>
<td>58</td>
</tr>
<tr>
<td>5.2</td>
<td>A usage of virtual ports that propagates backwards for values</td>
<td>59</td>
</tr>
<tr>
<td>5.3</td>
<td>The virtual ports of Figure 5.2</td>
<td>60</td>
</tr>
<tr>
<td>5.4</td>
<td>An implementation of the extended example that uses three virtual ports</td>
<td>61</td>
</tr>
<tr>
<td>5.5</td>
<td>The virtual ports of the extended example</td>
<td>62</td>
</tr>
<tr>
<td>5.6</td>
<td>An exemplary definition of two virtual ports</td>
<td>63</td>
</tr>
<tr>
<td>5.7</td>
<td>The virtual events of Figure 5.6</td>
<td>63</td>
</tr>
<tr>
<td>5.8</td>
<td>An implementation of the extended example that uses two virtual events</td>
<td>64</td>
</tr>
<tr>
<td>5.9</td>
<td>The virtual events of the extended example</td>
<td>64</td>
</tr>
<tr>
<td>6.1</td>
<td>The software interactions of the extended example</td>
<td>70</td>
</tr>
<tr>
<td>6.2</td>
<td>A MultiActivity with two threads</td>
<td>71</td>
</tr>
<tr>
<td>6.3</td>
<td>Exemplary definitions of group scripts</td>
<td>71</td>
</tr>
<tr>
<td>6.4</td>
<td>A script that moves a ball between three poles</td>
<td>72</td>
</tr>
<tr>
<td>6.5</td>
<td>The finite state automaton of Figure 6.4</td>
<td>73</td>
</tr>
<tr>
<td>6.6</td>
<td>A visualization of the script in Figure 6.4. The numbers adjacent to the software interactions correspond to the states of the FiniteStateAct of Figure 6.5</td>
<td>73</td>
</tr>
<tr>
<td>6.7</td>
<td>A script that moves information between SINK and SOURCE via QUEUE.</td>
<td>75</td>
</tr>
<tr>
<td>6.8</td>
<td>The four phases of Figure 6.7</td>
<td>75</td>
</tr>
<tr>
<td>6.9</td>
<td>An implementation of the extended example that applies two scripts</td>
<td>76</td>
</tr>
<tr>
<td>6.10</td>
<td>The aggregation of models that forms a multi-model script</td>
<td>79</td>
</tr>
<tr>
<td>6.11</td>
<td>An assembly for producing a helical motion</td>
<td>80</td>
</tr>
<tr>
<td>6.12</td>
<td>The assembly of Figure 6.11</td>
<td>81</td>
</tr>
<tr>
<td>6.13</td>
<td>A script using the assembly of Figure 6.11 to move a ball towards a pole helically</td>
<td>81</td>
</tr>
<tr>
<td>6.14</td>
<td>A hierarchical assembly</td>
<td>82</td>
</tr>
<tr>
<td>6.15</td>
<td>An assembly formed from the script of the extended example</td>
<td>83</td>
</tr>
<tr>
<td>6.16</td>
<td>The assembly of Figure 6.15</td>
<td>83</td>
</tr>
<tr>
<td>6.17</td>
<td>A usage of the assembly for the extended example</td>
<td>84</td>
</tr>
<tr>
<td>6.18</td>
<td>Complete Listing of Code for Extended Example</td>
<td>85</td>
</tr>
<tr>
<td>7.1</td>
<td>The three graphs of the general specification</td>
<td>87</td>
</tr>
</tbody>
</table>
11.10 Samples of applications built with RASP

11.11 Adapted applications with RASP

13.1 The three layers of Interaction-Based Simulation

A.1 An implementation in CORY
A.2 An implementation in CORY
A.3 An implementation in MAM/VRS
A.4 An implementation in FRAN
A.5 An implementation in ASAS
A.6 An implementation in RASP
A.7 An implementation in POLKA
A.8 An implementation in OBLIQ-3D
A.9 An implementation in RASP
Acknowledgements

The ideas and words of this thesis are my own, but none of this thesis would have been possible without the help and support of a great many individuals. To these people, I owe much thanks.

First and foremost, I thank my advisor, Gail Murphy. She is a great teacher and software engineer: dedicated, hard-working, yet compassionate and kind. Even while she labored with newborn twins, she continually challenged me to improve the quality of my work, the clarity of my thoughts, and the presentation of my ideas. I am thrilled to have met Gail, and it is an honor to have her as my mentor and friend.

I thank my supervisory committee (Dinesh Pai and Alan Mackworth of UBC, and David Forsey of Radical Entertainment) and my external examiners (Mark Greenstreet and Dale Cherchas of UBC, and Giancarlo Succi of the University of Alberta) for reading this thesis and providing many useful comments on content and presentation. I am especially grateful to Dave Forsey for sparking my interest in the subject matter and for serving as my first advisor at UBC.

I also thank the past and present graduate students and faculty of the Imager Laboratory of Computer Graphics and the Software Engineering Research Group. With these people, I spent countless numbers of days in conversation, laughter, and bewilderment. I owe special thanks to Robert Walker, a co-member of both labs, for taking interest in my work and commenting on my ideas.

I am grateful to the faculty and staff of the Department of Computer Science and of CICSР for providing me with financial support and granting me the opportunity to teach. One learns much as one attempts to mold the minds of the many. I am especially grateful to Kirsty Barclay, the CICSР reader, for proof-reading my thesis and providing helpful comments on grammar and presentation.

I am grateful to my close friends: Jason Smolensky, Jeff Sember, Danae Slater, Torre Zuk, Gretchen Greer, and Dave Morales. Over the years, they have been wonderful friends and a constant source of support. Many good meals were eaten together at The Great Wall, Mongolie BBQ.

I thank my sister, Gia Lee, for her kindness and her wisdom. Her endless words of encouragement and her commitment to helping me finish gave me the strength to complete my studies. I thank her for proof-reading my writing — a seemingly endless stream of text — and for treating me to a wonderful vacation to Europe — a timely reward for the completion
of this thesis.

Finally, I give my greatest thanks to my parents, Mansuk and Kyungja Lee, for their patience and moral guidance. Despite the years and a great physical distance, I always felt their presence and thoughts at my side and in my heart.

GENE S. LEE

The University of British Columbia
July 2001
To my parents and to the memories of my grandparents
Chapter 1

Introduction

The number and variety of applications employing techniques from computer graphics are growing rapidly. Traditional fields, such as computer animation and scientific visualization, and emerging fields, such as simulation visualization [38, 123, 157] and virtual reality [62], rely upon computer graphics to visualize the properties of dynamic systems. All of these fields produce *visual simulations*, computer simulations with visual properties. More specifically, visual simulations are computer simulations that visualize (in two- or three-dimensions) the experimental results of a *time-varying system*, a system that employs numerical modeling methods, either deterministic or stochastic, to update its state to the advancement of simulation time. Visual simulation is useful in that it serves as an effective medium for expressing scientific results, studying the behaviors of complex systems, and simulating the appearance of virtual worlds.

To meet the growing demand for visual simulation, academics and professionals are continually inventing new approaches to visual simulation so as to simplify their development and improve their performance. These efforts have produced faster hardware, better development tools, and superior design methods. Faster hardware facilitates visual simulations of greater complexity, increasing speeds, and enhanced visual displays. For example, Reality Engine [4], a high performance workstation, is an effective device for producing an immersive virtual environment. Better development tools improve the quality and accuracy of a visual simulation while decreasing developmental costs. Exemplary tools include VRML [159] and JAVA3D [33], both of which streamline the development of interactive applications. Superior design methods support the reuse of software, developed as components or scripts, and help manage software complexity. For instance, nodekits [145], abstractions that encapsulate an ordering of nodes, are useful for reusing code and controlling its complexity so as to produce visual simulations of moderate scale and size. The nodekits are applied like
CHAPTER 1. INTRODUCTION

templates to re-create useful motions and geometries.

1.1 Difficulties of Creating Visual Simulations

Despite the emerging popularity of visual simulations and the numerous advances in software and hardware supporting their development, the production of a visual simulation remains a difficult endeavor, requiring much time and effort. From its inception to its finish, a visual simulation may require many hours, days, or even years of work to produce. The costs of producing visual simulations, such as the animations of Toy Story [30], or of buying development packages for visual simulation, such as Alias|wavefront's MAYA [6], are high due in part to the time and expertise required to implement them.

Visual simulations are difficult to implement, especially with existing technologies, for three key reasons. First, visual simulations vary significantly in design and application. There is no one single animating method that models effectively the complexity of all visual simulations. For instance, some simulations, such as those that visualize the execution of an algorithm, are driven with high-level expressions, while others, such as those that animate the movements of an articulated figure, apply low-level instructions to set explicitly state changes over time.

Second, the development of visual simulations involves the use of advanced programming concepts, which include the meticulous advancement of simulation time, the execution of parallel behaviors, and the continuous update of state information. Simulation time synchronizes the global behavior of a system. It determines when a system changes state and in what order a succession of state changes occurs. Parallel behaviors enact simultaneous changes to a system's state with the advancement of simulation time. They induce concurrent computations and concurrent dataflow. State information determines the future behavior of a system. It requires continuous update to evolve properly and to produce accurate results.

Third, visual simulations are unlike traditional simulations, which develop systems that rely heavily upon their initial conditions and the occurrence of random events. Visual simulations are unusual in that they frequently apply controls and application frameworks to guide specifically the evolution of state values over time. The results of visual simulations are commonly known before they begin. In some cases, such as in the positioning of articulated figures [155], the aim of a visual simulation is to recover the proper initial conditions from a given result. For visual simulation, the methods of traditional simulation are useful in introducing abstractions and validating mathematical models that stress computation, not control or aesthetics.
CHAPTER 1. INTRODUCTION

1.2 Overview of Existing Approaches

The tools of visual simulation vary widely, as visual simulations are difficult to produce and the predominant tools for and approaches to building them rely upon conventional programming languages and design methodologies. Popular tools, such as OPENGL [105] and JAVA-3D [33], focus their efforts in defining and manipulating the shape and appearance of geometric entities. They facilitate visualization but provide few means for describing and coordinating the time-varying behaviors of the entities.

Tools that animate, such as TBAG [42] and OBLIQ-3D [104], tend to fix the architecture of a visual simulation to a single application framework. From the start of a visual simulation to its end, one framework decides how animation occurs and what techniques may be applied to animate state. For instance, the application framework of ASAS [127] supports processes, in contrast with the works of Fiume [47] and Kalra [74], which support temporal relations [7]. Each of these abstractions help manage the behavior and performance of visual simulations at a high level of abstraction. Other animating tools, such as GROOP [84] and UGA [172], provide a less restrictive approach to visual simulation by expanding their capabilities with special mechanisms, such as hooks and callbacks. These mechanisms permit developers of visual simulations to introduce new code, such as new animating methods. The mechanisms augment the functionality of the tools and vary the means for updating state. In GROOP, for instance, user-defined objects animate by inheriting functions that access and increment a global clock.

Be it for modeling or animation, the predominant approaches to visual simulation, such as those offered by INVENTOR [145] and ALICE [114], focus on producing prototypes. They neglect the concerns of software reuse and problems of growing complexity in favor of simplifying the rapid development and testing of new algorithms and interactive techniques. They provide abstractions that are often optimally defined for a single purpose. For instance, tools such as SWAMP [16] and CORY [56], animate state with only keyframing, an explicit mapping between time and state, while others such as ASAS [127] and OBLIQ-3D [104], animate state primarily with animated types, special variables that change state over time.

1.3 Shortcomings of Existing Approaches

Coding visual simulations with existing approaches is marked by three major shortcomings. First, existing approaches generally do not allow for the use and integration of several animating methods. The intricacies of visual simulation compound when multiple animating
methods are made available to animate state. Hence, it is not uncommon for the tools of visual simulation to support only a few animating methods. The tools limit developers from selecting the animating methods that work best for their visual simulations. Second, existing approaches do not provide for the effective management of software complexity. Too often, the concerns of rapid prototyping overshadow the concerns of building manageable software. Practical approaches, such as abstraction building and architectural design, are either missing or difficult to apply. The developers of visual simulations are left to handle complexity by their own means, leading to inconsistency among applications. Third, existing approaches do not support the benefits of software reuse. Building reusable software requires methodological planning; current approaches to visual simulation do not plan for reuse. Without an effective plan for reuse, visual simulations are difficult to decompose into reusable parts for sharing and improving the design of the simulations.

1.3.1 Animating Methods

Not every visual simulation animates well with the same animating method. For instance, one kind of visual simulation may animate well with keyframing, the pairing of key states with simulation time, while another is better suited to animate well with constraints, the building of dependencies between states. Keyframing fixes the state of animation from beginning to end while constraints adapt the state of animation to run-time events, such as the collision of moving objects. Other kinds of visual simulations may best operate by combining the two approaches and applying them simultaneously. The two methods may act independently upon different scenes, or both may work together to produce cumulative effects. Hence, it is beneficial for developers to be able to select and apply the methods that best match the needs of individual scenes. For instance, for an animation involving a camera that views a ball moving along a path, the camera's orientation is best set by placing constraints between it and the ball, and the ball's movements are best performed with keyframing.

Since general tools to mix animating methods are unavailable, developers exercise great care when using the tools of visual simulation, as noted by Green [62]. They carefully scrutinize their options and apply the tool (or tools) that best matches their needs. Once an appropriate tool is selected, developers produce visual simulations either by applying the tool and its animating methods directly to the system undergoing development, or by adapting the system so as to accommodate the tool. Both approaches produce adequate results but not without drawbacks. The former approach leads to inefficiencies and errors when the tool and its animating methods are not applied effectively. For example, keyframing
is an inappropriate method for animating a liquid. As noted by Kass and Miller [77], the movements of a liquid are non-linear and irreproducible with keyframes alone. The number of keyframes required to produce reasonable results is excessive and tedious to apply. The latter approach, adapting a system to accommodate the tool that builds it, frequently diminishes the accuracy of a visual simulation. When a system accommodates an external influence, such as the tool, it may adapt in a way that its design becomes overly complicated or unduly simplified. As suggested by many simulationists [169], the potential for error in such systems is large enough to throw into question the system's validity and to challenge the system's results. For example, if a liquid were animated as a few spheres to accommodate the use of fewer keyframes, its motion would probably be poor and suspect to misinterpretation.

Given such drawbacks, it is not uncommon for developers to forgo the tools of visual simulation for tools that create only shape and appearance, such as OpenGL [105] and QuickDraw-3D [11]. The developers animate their visual simulation by writing code specific to the needs of individual applications. Each application applies the animating method that seems best. That option provides developers with great flexibility, but its usefulness is limited. The coding process revolves around the development of individual applications: each application is written optimally to satisfy a single task. This approach hinders software reuse, or hinders the application of a new or alternative animating method. These drawbacks result in higher costs and longer development times.

In addition, computer simulation generally applies only one of several modeling techniques, such as those involving processes [90, 130], to build a complete system. That limitation, also characteristic of the many tools of visual simulation [84, 104, 42], precludes the production of systems that intermix multiple modeling techniques. Recent work in multi-modeling [46], the building of simulations with multiple modeling methods, indicates that intermixing is possible and a great benefit to simulation design. For visual simulations, the multi-modeling approach demonstrates the utility of intermixing the many methods for animating state and the many means for synchronizing parallel behaviors.

1.3.2 Complexity Management

With current approaches, as a visual simulation grows in size, it becomes more difficult to manage its complexity and make changes to its design. Much of the research in software engineering oriented towards managing complexity, such as framework design [20] and design patterns [51], has not yet been applied directly to the problems of visual simulation, in part because current approaches favor the building of prototypes. For instance, the procedural callbacks of DORE [85] and the animated types of MIRA [94] do not enable developers to
structure a visual simulation into representative parts and then determine how the parts assemble. Developers are left with the tasks of managing the complexity of code produced in these approaches and of ensuring that the approach they take is applied consistently. Unfortunately, consistency is difficult to maintain without a proper understanding of the growing complexities of a visual simulation and of how the complexities are handled. An approach applied inconsistently produces visual simulations that are difficult to build and troublesome to integrate. For example, the integration of MAM/VRS [36] with Anigraph [24] does not occur easily although each applies a “graph-based” approach to temporal sequencing. Operating dissimilarly, the graphs of MAM sequence the execution of behaviors while the graphs of Anigraph sequence the continuity of dataflow.

A further problem that may arise when developers create their own approaches to handle complexity is architectural mismatch, the use of software in contexts that mismatch those intended by its creators [53]. In basing the structure and assembly of their programs on mismatched assumptions, developers, for example, may use incompatible control loops or may combine inappropriate parts. Architectural mismatch can produce code susceptible to error and sensitive to design changes. Errors result, for instance, if the low level primitives of Clockworks [56] are altered by controls that neglect to communicate their changes to Clockworks’s high level abstractions, “cues” and “scenes.” Unaware of any changes caused by the controls, the high-level abstractions use stale information to compute faulty results. Design changes, such as those changing the timing of visual simulations, cause components to compute erroneously if the components relate their computations to a particular time or timestep.

To avoid the difficulties of managing complexity, some developers may choose either to resolve such problems as they occur, or to devise separate approaches for each visual simulation. Although both options can lead to quick results, each provides only short-term gains. The first option manages the overall complexity of a visual simulation with only local solutions while the second choice supports inconsistencies in the development of visual simulation. With regard to the former, local solutions remedy a small problem without consideration of their global effects on other visual simulations. Thus, this approach does not take into account how a visual simulation is formed from parts or how the parts cooperate to perform specific tasks, limiting a developer’s ability to secure long-term gains. For example, the application of inheritance and templates to animation languages, such as Charli [29] and TBAG [42], avoids the replication of code but provides no assistance in assembling the overall design of a visual simulation. As noted by the proponents of framework design [72, 119, 20], a proper understanding of a program and its assembly is essential to manage the complexity
of a system and to understand its design.

With regard to the latter option of devising separates approaches for each visual simulation, the inconsistencies that result seriously hamper the effective development of complex visual simulations. The parts of a system and their assembly ultimately determine what a visual simulation computes and how animation occurs. Each part performs an important role in determining what a visual simulation can do easily and what it cannot do without difficulty. If the design and use of parts are inconsistent or varying across applications, developers are unable to apply changes to visual simulations readily, especially changes concerning who is animating, what animation is occurring, and when the animation performs. A varying use of parts, moreover, requires practitioners to learn the variants and to recall which parts of each visual simulation implement its variations. Ultimately, that requirement diminishes the efficacy of software development and amplifies the difficulties of software complexity. For example, inheritance differs greatly from delegation when applied to manage the complexities of visual simulation. Inheritance, as applied by MAM [36], encapsulates behaviors within separate classes while delegation, as applied dynamically by UGA [172], introduces behaviors to class instances with time-varying messages.

1.3.3 Software Reuse

The reuse of a visual simulation or its parts is not an easy task. Few tools of visual simulation directly apply the popular approaches of software reuse, such as component software [1] or domain engineering [106], to the reuse of a visual simulation and its parts. In preparing the development of prototypes, most tools ignore the concerns of generality and adaptability, two important forms of software reuse [59]. The few that do, such as VRML [159] and OPENGL [105], are effective only in developing and reusing the parts of a visual simulation that form and render geometric shapes. They provide developers with few means to reuse the parts of a visual simulation that animate. Developers of visual simulation must decide independently how to separate their animation into parts and to re-apply them elsewhere. Unfortunately, experience [163, 19, 148] shows that such an approach to software reuse rarely succeeds. Effective reuse requires the use of an explicit agenda for software reuse and a systematic reuse-directed software process [71].

Lacking tools to support reuse, developers encounter many difficulties in attempting to separate their visual simulations into reusable parts. Some tools and approaches to visual simulation establish bindings that are useful only to the needs of specific applications. The bindings fix the order and times in which the many parts of a program communicate. In doing so, the bindings complicate the design of the parts and hamper the general applicability.
of the parts to a wider variety of applications. To reuse parts effectively, developers must extract or alter the bindings so as to remove unwanted communications. Developers do so, for example, with ASAS [127] and POLKA [143]. Each tool contains numerous elements that fix the communications between their parts. Some other tools and approaches of visual simulation tightly integrate the statements that control the interaction and behavior of their parts. The separation of these statements is difficult and frequently requires the parts to undergo design changes when applied to new applications. The approaches of FRAN [41] and UGA [172], for example, intertwine the parts of visual simulations that update state with those that manage time.

Beyond the difficulty of separating a visual simulation into its representative parts, problems also arise because the parts of a visual simulation rarely accommodate reuse in alternative contexts. The contexts in which the parts operate may vary widely from one simulation to another. In one context, the interactions of the parts may vary while in another, the parts’ operation may change over time. Not designed for reuse, the parts rarely adapt well to evolutionary changes, such as the application of animation techniques with simulation modeling. Animation techniques, such as keyframes [16, 56] and animated types [127, 104], form diverse mappings between time and state. Simulation modeling, including functional modeling [136, 73] and constraint methods [117], can apply such mappings to build complex systems. Further troubles arise if the parts heavily rely upon one specific context. For example, the parts of a visual simulation that require constant updates over time, such as those that increment a positional value by a constant for a set interval of time, commonly falter when integrated into an environment that updates time unevenly. The parts incorrectly perceive the passage of time as constant and thereby produce inconsistent results, as exemplified by the advancement of the positional value with inaccurate increments.

In light of the difficulties of applying software reuse and the troubles of securing reuse from tools that primarily support prototype development, it is not uncommon for developers to overlook the process of engaging in software reuse. Instead, they opt to produce, or assemble, their visual simulations by coding each application anew. Avoiding the obstacles of software reuse, they accept the difficulties accompanying the development of new code, such as design flaws and coding errors.

1.4 Thesis Statement

This thesis shows that high-quality visual simulations are easier to devise, manage, and reuse when interactions between software components that make up visual simulations are
given first-class status. The first-class interactions — *software interactions* — manage and regulate the communications between software components by deciding when and how the components transmit information. As illustrated in Figure 1.1, software interactions liberate the components from managing their own communications. The components act without acknowledging the identities of their communicating partners or coupling tightly with one another. Thus, the components are free from administering to the needs of complex communications and are better equipped to accommodate their various complexities. The software interactions also act as intermediaries for the transmission and manipulation of information. They adapt communication so as to ensure that all components send and receive information properly. In the long run, software interactions encourage the production of programs by composition and facilitate the extraction and application of software components for reuse and adaptation. To demonstrate the utility of software interactions for building visual simulations, this thesis introduces *interaction-based simulation* (IBS), a new approach to developing time-varying systems. An implementation of IBS is presented in the form of the RASP toolkit, a software library of stylized C++.

![Figure 1.1: Three software components under the influence of two software interactions](image)

### 1.4.1 Interaction-Based Simulation

As shown in Figure 1.2, IBS consists of five basic parts, which are organized in three layers. Layer one forms the basis of IBS with *interaction-based programming* (IBP), a new approach to program development that works with software interactions. Layer two augments the first layer with *interaction constraints, multi-modeling methods,* and a *general specification*. Together, the three parts apply IBP to building visual simulations. Interaction constraints and multi-modeling methods apply the methods of constraint technology [132] and computer simulation to the application of software interactions, while the general specification partitions visual simulations into the product of three graphs: the first graph for assembling parts, the second for organizing software interactions, and the third for deciding time-varying behaviors. Layer three provides structure to the application of the first two layers. It presents *reuse guidelines*, software properties for preparing reusable software.
Interaction-Based Programming

IBP is an approach to programming that supports the building and reuse of software by parts. An IBP program is made up of two parts, *activities for production* and *activities for coordination*, which are developed separately. The former activities identify a program by its parts; the latter manage a program by its communications. The two activities act collectively to produce a program that continually mediates the flow of information among its parts.

The activities for production consist of software components, which establish state and transform input to output, and software interactions, which identify the communications among components. As shown in Figure 1.3, the two constituents establish a *software network* that defines the topology of the program and decides its execution. When the network operates, by way of the activities for coordination and a run-time evaluator (RTE), the software interactions force the software components to communicate their state and to react to their communications.

The activities for coordination produce and manage the execution of software interactions. As shown in Figure 1.4, they consist of two kinds of abstractions, *scripts* and *meta-scripts*. Scripts configure an arrangement and ordering of software interactions so as
to produce a specific sequence of communications. These communications decide what a program computes and ultimately, how it animates its state. Meta-scripts decide how the RTE executes individual scripts or how the RTE switches its focus from one meta-script to the next. They present rules, statements, and conditions for producing a program as a sequence of scripts for controlling the use of software interactions. Meta-scripts support animation by relating the execution of multiple scripts to the value of simulation time. As time flows, the meta-scripts indicate which scripts execute in sequence and which scripts execute in parallel. For example, the meta-script of Figure 1.4 indicates that script X precedes the parallel executions of scripts Y and Z.

```
Meta-Script
AT T=100:
   ScriptX
AT T=200:
   ScriptY + ScriptZ
AT T=300:
   ScriptZ
```

Figure 1.4: The activities for coordination consist of scripts that determine the execution of a software network.

As the RTE sequences through several scripts, the topology of a program changes. As shown in Figure 1.5, the progress of simulation time updates the configuration of a visual simulation to include new connections or new components. Unlike the process in conventional programming approaches, the couplings between components in an IBP program are neither fixed nor always deterministic. The program may dynamically switch from one software network to another, permitting a visual simulation to vary radically its behavior and its use of animating methods with the passage of time.

```
Sequence of ScriptX
Sequence of ScriptY
Sequence of ScriptK
```

Figure 1.5: As individual scripts execute, they produce topological changes to the interconnections amongst software components.
Interaction Constraints

Interaction constraints apply constraint technology to the application and use of software interactions. Constraint technology, such as unidirectional constraints with forward propagation, coordinate complex communications with *equations of interactions*. The equations, established with mathematical operators and primitive functions, determine how software interactions relate and disseminate information. For example, the interaction constraint in Figure 1.6 establishes VP, a virtual port that permits interactionB to communicate the sum of compA and compB. Each time interactionB requires a value, VP applies the operator + to the two components.

![Figure 1.6: An interaction constraint that relates the sum of compA and compB.](image)

Multi-Modeling Methods

Multi-modeling methods support the development and reuse of *multi-model simulation*, a simulation composed of multifarious subsystems. Each subsystem employs a design method for organizing software interactions that best matches the problem it models. As noted by Fishwick [46], the subsystems collaborate to produce a simulation of greater accuracy and simpler design. The design of a simulation and its mapping of variables to animating methods are chosen by the developers, not the tools employed by the developer. The multi-model simulation of Figure 1.7, for example, combines several design methodologies into one system. The primary design method “functional” partitions the simulation into five subsystems, with data flowing sequentially from one subsystem to the next. Each subsystem, in turn, applies a different method for mediating the software interaction of various animating methods. Subsystems Q and S, for instance, apply the methods “declarative” and “constraint” respectively.
CHAPTER 1. INTRODUCTION

General Specification

The general specification establishes an extensible means for describing the contents and behaviors of a visual simulation. Supporting the development of software by composition, it partitions a visual simulation into three graphs: scene, action, and time. The scene graph assembles software components to define geometric shapes with physical attributes; the action graph applies interaction constraints and multi-modeling methods to produce scripts; and the time graph uses temporal operators to produce meta-scripts. The temporal operators determine when the scripts of the action graph communicate information to and from the shapes of the scene graph. As shown in Figure 1.8, the three graphs establish a logical approach to constructing a visual simulation incrementally. Developers simply add (and remove) nodes from the graphs to vary or change a program's complexity or behavior.

Reuse Guidelines

Reuse guidelines improve the reusability of software components involved with software interactions. The guidelines are divided into three sets of properties: interactional, behavioral, and temporal. The properties determine how software components communicate information.
CHAPTER 1. INTRODUCTION

Inter action, complete computational tasks, and behave over time. The guidelines ensure that the use of software components is consistent with the foundations of IBP: software components should interact freely with software interactions and with the environments that software interactions create. When software components operate consistently, they are simpler to apply in varying contexts, especially those that change their communications over time.

1.4.2 Benefits of Interaction-Based Simulation

The benefits of IBS are four-fold: it provides developers with a programming approach that eases the production of visual simulations, a means to integrate multiple animating methods, a basis to manage complexity, and a means to apply software reuse.

Programming Approach

IBS eases the task of programming time-varying systems. Its design separates the specification of communications from its execution. With the separation, developers can update and abstract the communications of a program without preparing controls for its execution. The execution of a program varies according to its specification, not to the ordering or sequencing of function calls, or of operational controls supporting its specification. Hence, the specification of an IBS program contains no elements, such as semaphores, that apply to the execution of communications nor reference statements, such as co-routines, that cause its underlying program to switch between multiple contexts. The absence of these types of commands, which unnecessarily obfuscate the meaning and application of a program’s communications, is a significant benefit of IBS.

Multiple Animating Methods

IBS eases the application and integration of multiple animating methods. Activities of coordination structure a program as a collection of software networks, each reproducing the communications of a particular animating method. Animation occurs as the software networks communicate the advancement of simulation time to its components. The animation varies its methods as software networks vary their communications and act in parallel. The use of multi-modeling methods allows for practical means to vary and to integrate the animating methods of several software networks. Applied with controls for relating the networks over time, the multi-modeling methods either embed the networks hierarchically or link the networks in sequence.
CHAPTER 1. INTRODUCTION

Complexity Management

IBS introduces three means to ease the task of managing the complexity of visual simulations. First, it partitions a program into two parts. One part establishes computational activities while the other organizes communicational activities. The two parts act independently, each providing an alternative means for managing software complexity. Second, IBS supports the use of interaction constraints so as to simplify the application and execution of software networks. The interaction constraints reduce the number and complexity of software interactions in order to clearly represent complex communications. Third, IBS encourages the development of a general specification for visual simulation. Application development involves the building of software via software composition, the building of code with abstractions that generalize objects, agents, and functions. Using the general specification, developers methodically organize software interactions into layers, orderly arrangements, and recursive structures.

Software Reuse

IBS facilitates the reuse of software by producing programs with IBP and by advocating the use of reuse guidelines. With IBP, computational components operate without direct reference to their environment. Therefore, they are easier to extract and to apply in new applications. The reuse guidelines ensure the consistency of components as they operate in varying contexts. With IBP, the context of components change as they either encounter new software interactions over time or operate in new applications.

1.5 Thesis Overview

The remainder of this thesis consists of twelve chapters. Chapter 2 provides an overview of previous work. It discusses the strengths and drawbacks of various tools for visual simulation that are commonly used in the fields of computer graphics and computer simulation. Chapter 3 provides an overview of IBS and introduces a general example for presenting its concepts. Chapters 4 through 8 describe the five major parts of IBS: interaction-based programming, interaction constraints, multi-modeling methods, the general specification, and reuse guidelines. Each of the five chapters discusses the details of one part of IBS and demonstrates its application through further elaboration of the general example set forth in chapter 3. Chapters 9 through 11 provide evidence to the claims of this thesis. These chapters show how IBS is better enabled than existing methods and tools to support the complexities of
developing visual simulations of wide variety and kind. The first two chapters (9 and 10) use empirical results and logical arguments to validate the design of IBS. Each of these two chapters presents a set of examples and directly compares an implementation of the examples using IBS to implementations using the currently available visual simulation tools. The examples are representative of common scenarios appearing in visual simulations. The tools chosen for comparison are representative of the currently available approaches to visual simulation. The comparisons demonstrate that IBS is capable of expressing common scenarios with additional desirable properties in comparison to existing approaches. The third chapter (11) assesses the utility of IBS in easing the development of visual simulations. It presents logical arguments to describe how IBS simplifies the development of visual simulations and presents qualitative results from the use of IBS that support the claims of this thesis. Chapter 12 compares IBS to other approaches in software engineering, computer graphics, and computer simulation. The software interactions presented here share some similarities with those in previous works, but none of those works apply software interactions in precisely the same way. Chapter 13 summarizes this thesis and presents future work. It also details the contributions of the thesis to the development of visual simulations.
Chapter 2

Background

The literature abounds with assorted descriptions of development tools for visual simulation. Practically all of these tools originate in the fields of computer graphics and computer simulation. Earlier tools from these fields were useful but limited. Those from computer graphics primarily visualized information while those from computer simulation primarily organized state changes over time. More recent tools from each field attempt to do both. Computer graphics employs simulation techniques to animate its scenes while computer simulation employs visualization techniques to animate its execution.

Despite the convergence of the two fields, each field produces visual simulations for different purposes. The computer graphics community creates guided simulations while the computer simulation community creates undirected simulations. Guided simulations manage information to reproduce deterministic results. Their outputs are generally known before they begin. Undirected simulations manage information to present or compute results. Their outputs are not usually known before they begin and are discovered only after they end. Modern computer games blend both graphics and simulation. Most of the play of a game is guided; secondary motions, which appear in the background or produce ornamental effects, are developed as an undirected simulation. This approach is taken to ensure the outputs of a game are restricted to those that produce "interesting" results.

This chapter discusses briefly the designs of previous development tools for visual simulation. For both computer graphics and computer simulation, the discussion presents a structural overview of the tools and the software approaches the tools enlist to establish and to control dynamic systems. It explains how the separate aims of the two fields encourage differences in the designs of their tools and in the software engineering techniques they employ. The chapter concludes with a short commentary on the role of communications in software programs. It discusses how the tools of visual simulation provide features for its
control.

2.1 Computer Animation

The computer graphics community produces computer animations – plausible, visual simulations. The animations emphasize aesthetics, interaction, and control over computation and analysis. The outcome of the animations are usually known before they begin.\(^1\) Precision and accuracy are desirable goals but not absolutely essential. Imprecise movements and exaggerated motions are often acceptable if they convey reasonable results. The aim of visualization is to present the results of controlled computation. As computations occur, visuals animate. Each time step in a computer animation updates the values effecting computation. For instance, a free-falling object’s position updates as time progresses, and an articulated figure grabs as a planning algorithm calculates. For many computer animations, there exists a simple mapping between state variables and visual attributes. As the variables update, the visual attributes change.

2.1.1 Structural Overview

Emulating the steps of traditional hand-drawn animation, the graphics community decomposes computer animations into three production phases: modeling, rendering, and animating. Modeling involves the building of virtual scenes with lights, cameras, and shapes. Rendering entails the interpretation of virtual scenes to yield synthetic imagery. Animation involves the changing of virtual scenes over time, as when animating the entities of modeling and the visuals of rendering. The interplay of the three phases produces a series of discrete images that visualize the contents and behaviors of virtual scenes.

Many tools for graphics, such as GROOP [84] and GRAMS [40], address only a subset of the three phases. They offer primary support for one or two phases and secondary support for the others that remain. In building a complete application, the tools expect developers to introduce additional tools for managing the phases that are incomplete. Developers either produce the additional tools themselves or find other tools to address their needs. Secondary support often comes in the form of callbacks, functions, or extensions. Developers apply these three forms in integrating their programs with the tools. All three forms enable developers to access and exchange information with the tools so as to accommodate the

\(^1\)Unpredictable results may arise with keyframing in the form of secondary effects. The overall movements of an articulated figure, for instance, are unpredictable if each of its joints is animated simultaneously with keyframing.
CHAPTER 2. BACKGROUND

needs of specific applications. For example, VRML [159, 18], a specification for modeling, animates by providing hooks for scripting utilities. Developers apply the scripting utility of their choice in integrating the hooks with their code.²

Of the tools for graphics that address multiple phases, most aim to support only modeling and rendering. Animation is a secondary concern and often given minimal support, usually in the form of functions or interfaces. Functions, as provided by BAGS [144] and GRAMS [40], create specific effects, such as spinning an object around a point. Functions are simple to use but difficult to apply as computer animations increase in complexity. They frequently apply their effects directly to state variables and thus tend not to accommodate the application of their effects elsewhere, nor their collaboration with others in producing higher-order effects, such as spinning an object sporadically around a moving point. In addition, the functions are difficult to manage as the rising complexity of an animation produces a greater number of function calls. Few means are available to correlate their use or to apply them in forming higher-level abstractions.

Interfaces, as supported by TWIXT [61], UGA [172], and OBLIQUE-3D [104], provide hooks for developers to associate state variables to controllers, specialized components that regulate the evolution of state. As the controllers compute new states, the interfaces relay the changes to the state variables. This approach shares several similarities with the approach presented in this thesis, but there are several differences. IBS presents additional elements, such as scripts and meta-scripts, that compose and organize interactions between interfaces and controllers. Most tools supporting interfaces require developers manually to establish, manage, and control these interactions. IBS also employs time to control the dynamic properties of the interactions. Time produces communications, not state changes. State changes happen as communications occur.

2.1.2 Animating Methods

The tools for graphics that address animation do so directly by applying a variety of methods for animating state. Each method applies a different construct in establishing a mapping between state and time. The methods decide when the mappings change, and for many, what the mappings compute. The constructs which distinguish the methods are keyframes, messages, animated types, and declarative expressions. Keyframes are "time, value" pairs; messages are coded signals; animated types are self-modifying variables; and declarative expressions are specialized statements. Each of the constructs makes trade-offs between

²The scripting utilities often applied with VRML include Java [12] and ECMAScript [39].
CHAPTER 2. BACKGROUND

Keyframes are data pairs that form explicit mappings between fixed states and future times. The mapping establishes a low-level specification that requires most, if not all of its states to be known before an animation starts. Animation based on keyframes is referred to as event-based: a series of keyframes records the events that signal a change in state. Keyframes are attractive in that they are simple to create, quick to interpret, and easy to store. They manage state changes effectively and support numerous interpolation techniques [81, 168]. Animation systems that rely upon keyframes include SWAMP [15], TWIXT [61], and CLOCKWORKS [56]. The systems differ in their development of keyframes and in their means of enlisting keyframes to animate state. In developing keyframes, SWAMP produces them as "streams," TWIXT produces them as "tracks," and CLOCKWORKS produces them as "evaluators."

Despite their great utility, keyframes are not useful for producing all types of animation, especially those of higher complexity. Few primitives exist to manage the use of keyframes at multiple levels of abstraction. For instance, signal processing methods [168, 25] prove useful in adapting keyframes to new motions, but not in associating the keyframes with logical expressions or temporal controls. Keyframes are also not useful in encouraging a separation between state and time. Primitives are generally unavailable to adapt keyframes to run-time conditions, such as the activation of an event or the appearance of obstacles.

Messages are signals passing between the interfaces of components. The signals are useful in establishing communications, declaring state changes, and synchronizing temporal events. Messages are often associated with time stamps that determine when they appear and how they act. Animation systems using messages to animate state include ASAS [127] and UGA [172]. Messages are useful and simple to apply but arduous to handle as the complexities of an animation grow [166]. Few means are available to integrate the effects of multiple messages or to coordinate effectively the passing of many messages. In addition, rules do not exist to standardize the way that components interpret messages and react to communications. Such rules are important in improving the reuse of the components in multiple contexts and having the components work properly in time-varying systems.

Animated types are variables bound with time-varying structures. As the structures compute new values over time, the variables change state. Animated types range from simple interpolating variables, such as the Newtons of ASAS [127], to complex constraints, such as the behavioral functions of T-BAG [42] and OBLIQ-3D [104]. With animated types, the changing of state is usually automatic and fixed with time. Few mechanisms exist to regulate carefully the timing of state updates and to control dynamically the associations between
variables and time-varying structures. Of the few that do exist, such as the behaviors of T-BAG, most provide limited support for developing a wide variety of scenes.

Declarative expressions are statements detailing the contents of an animation and presenting its dynamic properties. Organized into a language, the expressions indicate "what" is happening and "when" it occurs. A specialized interpreter evaluates the expressions in deciding "how" an animation progresses over time. Most systems employing declarative expressions, such as Anigraph [24], Fran [41], Mam/Vrs [36], and Arya [13], organize the expressions into node graphs or recursive lists. The two data structures ease the use of applying expressions and support the development of software by composition. The drawbacks of applying declarative expressions are the same as those of applying a conventional programming language to animation. The expressions are not well-suited for specifying concurrent state changes and encouraging the reuse of high level abstractions. Most lack rules or guidelines for handling the intricacies of a complex system. Thus, as a system grows complex, the declarative expressions produce a large set of statements that are difficult to comprehend, debug, and reuse.

2.1.3 Complexity Management

With most tools for animation, the management of complexity is overshadowed by the goals of prototype design. The tools strive solely to simplify the development of animation with compact notations and simple structures. The most common approach to handle complexity uses extension mechanisms, such as templates and callbacks. Such mechanism not only introduce extensions, but also facilitate the management of a moderate growth in complexity. Other mechanisms providing utility include the use of constraints [42, 76, 58], lazy evaluation [14, 41], spatial operators [14], behavioral transformations [14], and object-oriented design [16, 29, 56, 172, 113].

While useful, extension mechanisms provide insufficient means to manage animations of great complexity. The mechanisms are unsuitable for manipulating animations at multiple levels of abstraction, a task providing innumerable benefits in dealing with complex systems. Computer simulation, for instance, introduces higher-order mechanisms, such as processes, to manage the intricacies of a simulation at a level higher than the handling of events and state changes. The extension mechanisms are also ineffective in introducing new animating methods, and certainly cannot alter the application of animating methods over time. The complexity of an animation grows dramatically if only one method, such as keyframing, is applied to reproduce the effects of other methods, such as those involving physically-based motions [152], particle systems [126], or algorithmic animation [143].
A number of tools in graphics, such as Frames [118] and Condor [76], manage the complexity of an animation with software composition, the methodical integration of software components that encapsulate functions and commands. Integration involves the organization of components into hierarchies, orderly arrangements, or nodes of a dataflow graph. An evaluator parses the organization of the components so as to build and execute a successful animation. The primary use of software composition, when applied by the tools, is the establishment of an ordering of state changes, or the development of a set of relations. The ordering of state changes, as exercised by Phigs [154] and GKS [69], models an animation at a low-level of abstraction. The state changes directly affect the values of specific states. The ordering of state changes neither establishes nor modifies the behaviors of higher-order abstractions. Conversely, the ordering of relations, as employed by Inventor [145] and Groop [84], models an animation at a high-level of abstraction. The ordering determines how the components relate individually or in groups. Most often, the relations that develop between components are fixed. It is difficult to introduce new relations that are not provided directly by the tools. Moreover, most relations among components are static. Few tools, not just those based on software composition, provide any means to augment the variety of existing relations or to manipulate these relations dynamically. This deficiency dramatically increases the complexity of an animation that attempts to vary these relations over time.

2.1.4 Software Reuse

Rapid development is a primary goal of the current tools of animation. Prototyping helps developers test and apply new algorithms or techniques to specific scenes or contexts. However, when developers wish to re-apply the same algorithms or techniques, such as those controlling the movement of a crowd or the collaboration between two figures, to a wider variety of scenes or to a greater number of contexts, the existing tools are limiting. Except for those employing declarative expressions, existing tools primarily offer low-level structures that are simple or compact. As the complexities of the animations grow, the animations and their parts often become difficult to manage and impractical to reuse.\(^3\) Tools using declarative expressions offer some utility in that they may apply practical approaches in software development, such as software composition, to the building of reusable code. But like those providing low-level structures, the tools using declarative structures are designed

---

\(^3\)Occasionally the term "reuse" is applied by the animation community with regard to the development of "reusable motion libraries" [168, 25] and "reusable motion controllers" [155]. Rather than addressing the reuse of code, these terms refer to the ability of an animation to apply and integrate the motions of an articulated figure or the streams of motion-captured data.
for building prototypes, not reusable applications.

As such, a specific agenda of reuse is unsupported with the tools of animation. Some tools, such as VRML, accommodate reuse for modeling and rendering, but not for animation. This is partly because the processes of modeling and rendering are better understood. In several tools, especially those applying software composition, there is an implicit suggestion of reuse. Components that compose well should decompose easily into its representative parts. This approach works well if the parts that developers wish to reuse do not involve cross-cutting concerns. Except for a few, such as ARYA [14] and ASADAL/PROTO [80], the tools of animation do not propose a specification that controls independently the “who,” “what,” and “when” of a simulation. In facilitating prototype design, most specifications concisely indicate who changes, in what way they change, and when the changes occur. It is difficult to partition the specification and reuse its parts separately.

2.2 Computer Simulation

Unlike the computer graphics community, the computer simulation community creates visual simulations to verify, present, and validate mathematical models. They stress computation, not control or aesthetics. Simulations establish initial conditions, execute state changes, and monitor results. Unlike their counterparts in computer graphics, they observe, not control, the production of results. Simulations execute repeatedly to ensure that they observe the proper relationship between initial conditions and final results. Attempts to integrate general simulation techniques with computer graphics have produced mixed results. Many attempts, such as those based loosely on physics [152], serve limited use. It is difficult to predict and to vary the outcome of such simulations. However, some integrated approaches permit great flexibility. Simulations, such as those based on L-systems [124] or particle systems [126], provide adequate controls for users to carefully produce deterministic results.

Simulation research, unlike graphics research, attempts to analyze, classify, optimize, and verify a simulation’s design. From a simulation’s design, it is possible to identify the type of systems it can simulate, the degree of accuracy to which it performs, and the roles of its components. Recognition of the roles of components and their interactions is also important for software reuse and integration. It defines how components communicate, relate, and behave. As advocated by many [106], reuse of components is greatest when components

---

4To validate a model means to prove that it is an adequate representation of a system under investigation. To verify a model means to check the consistency of its design (i.e. accuracy, numerical solutions, etc.).
move to and from similar environments, undergo similar working constraints, and accept similar roles and responsibilities.

2.2.1 Structural Overview

The computer simulation community dissects a simulation into four phases: *model design*, *model execution*, *model verification*, and *data analysis* [119, 151, 46]. Model design produces the simulation; model execution executes the simulation; model verification conforms the validity of the simulation; and data analysis reviews the results of the simulation. As illustrated in Figure 2.1, all four phases are essential for building a reliable simulation. Model design is critical to the overall accuracy and usefulness of a simulation. An improperly formed design critically affects each of the remaining phases. Similarly, model verification is critical to assessing the reliability of a simulation. Thus, as indicated by the backwards facing arrowheads, simulation development is an iterative process. Each time a deficiency or problem is encountered, the four phases repeat.

Model design, the phase of utmost importance in developing visual simulations, consists of the development of entities, an executive logic, a temporal management strategy, and an approach to display building. Entities are the elements of primary interest within a simulation. They represent tangible things, such as machines and parts, or intangible things, such as queues and routes. The executive logic refers to the computational algorithms applied by a simulation for altering the states of its entities. As a simulation progresses, the executive logic controls the mapping of state to time. The temporal management strategy is the method applied by a simulation in determining when and how simulation time advances. The temporal management strategy affects when and how the executive logic acts. Display building is the process of visualizing a simulation for presentation and analysis. Most often, the visualization conveys the behaviors and attributes of the entities or of the executive logic.
CHAPTER 2. BACKGROUND

Classification

Many classifications of simulation [169] identify the temporal management strategy as the distinguishing characteristic. As noted in Figure 2.2, temporal management strategies divide into two types, discrete-time and continuous-time. In discrete-time simulation, time updates incrementally at fixed intervals. All state changes occur only at the times set by the incremental updates. In continuous-time simulation, time flows smoothly at varying intervals. All state changes occur freely and arise at any moment in time.\(^5\)

![Diagram of temporal management methods](image)

Figure 2.2: The methods of temporal management [169]

Of the two types, discrete-time simulation is simpler to build and is used frequently by the tools of computer graphics. A discrete-time simulation can be as simple as a program that iterates continuously over a set of functions. Each iteration embodies an incremental progression of time. The popularity of discrete-time simulation in graphics [84] is probably due to the nature of animation, which involves the periodic display of visual imagery. Unfortunately, discrete-time simulation is not apt in producing simulations of superior quality and size. It is not favored because it aligns state changes to fixed increments in time. A simulation is not permitted to reproduce accurately any event occurring in between time steps. Ultimately, this limits the worth of a simulation's results, and reduces the validity of its design. In addition, discrete-time simulation is not favored because it fixes the pace of a simulation to a constant rate. A simulation can not update faster than the rate of its slowest part. Otherwise, the simulation would suffer from temporal aliasing, the buildup of artifacts arising from a lack of temporal resolution.

Continuous-time simulations, which involve greater efforts to build, are of two types, continuous state and discrete-event. In continuous-state simulation, state variables undergo

\(^5\)In a literal sense, all computer simulations advance time discretely, since computers and programming languages behave discretely. However, it is the reaction of state variables to temporal advancement that separates the two simulation types.
smooth changes. Most often, the changes derive from the solutions of mathematical equations, which are either differential or algebraic. As exemplified by the tools CSMP [141] and DESIRE [83], continuous-state simulations are normally associated with block diagram methods for control and system engineering. In discrete-event simulation, state variables update discretely and simulation time jumps irregularly. The popular forms of discrete-event simulation are event scheduling, activity scanning, and process interaction. In event scheduling, simulation time advances according to the scheduling of events, the times of important state changes. Time advances incrementally from one event to the next. In activity scanning, time advances according to the evaluation of activities, sets of events paired with conditionals. Activities with truthful conditionals apply their events in advancing time. In process interaction, simulation time advances according to the performance of processes, tasks that encapsulate conditionals and events. As the processes interact, they compete for resources and schedule the timing of events. The tools of simulation that perform discrete-event simulation include GPSS [133], SLAM [122], and SIMSCRIPT [78].

2.2.2 Simulation Methods

The methods that the tools of simulation apply in developing simulation vary. Often, the system under investigation influences a simulation’s design and the representation of its parts. For example, Petri nets [116] easily model systems involving resource contention, while engineering block diagrams [137] easily model systems involving continuous variables. Therefore, the tools of simulation that support several methods are favored over those that support only one. Those that support only one are uni-model while those that support multiple methods, which may be either discrete-event or continuous-state, are multi-model. As noted by various researchers [109, 46, 171], multi-model tools support multiple approaches within one developmental framework. They provide a unified approach to simulation development that provides greater flexibility and superior design. They permit developers to partition simulations into parts and model each part with the approach that best emulates the part’s design and function.

Uni-model Approaches

Fishwick [46] classifies four uni-model approaches based upon the implementation and cooperation of the four parts of every simulation. The four approaches are declarative, functional, constraint, and spatial. Declarative approaches, as exemplified by transition functions [112] and event graphs [135], focus on the concept of state. For each state, there are
rules that identify when it occurs and when it changes. Collectively, the rules establish a total ordering of the state space. Each rule updates the system from one state to the next. Declarative approaches often employ transition tables to model this ordering of state.

Functional approaches, as employed by CSIM [136] and Sim++ [90], produce networks of coupled blocks. Data, representing functions or variables, flow through the network from one block to the next. As data flow, states update and events occur. Functional approaches best model systems in which a directionality of flow exists between system components, such as queues and control engineering. By contrast, constraint approaches model non-directional relationships, such as the laws of nature, with difference or differential equations. They update state by converting the equations into first-order forms, and then solving the forms with numerical integration techniques.

Spatial approaches, as employed by L-systems [124], decompose simulated environments into many basic elements acting in parallel. The properties of the system under investigation determine how decomposition occurs and how basic elements interact. The two most common decompositions divide environments into many spaces, such as a lattice of cellular automata, and into many entities, such as a branching network of L-systems.

**Multi-model Approaches**

Multi-model tools encourage developers to mix and match the subsystems of a simulation with the individual methods that best match a subsystem's behavior and structure. Architecturally, this mix and match process forms a network of subsystems that cooperate to solve a larger task.

The integration of uni-model approaches is done with one of two methods, *combination* or *refinement*. Combination methods present a unified approach to simulation modeling. As exemplified by GASP [122], this usually entails a combination of continuous state modeling with discrete-event operations. Refinement methods, such as those offered by Simpack [45], MAST [10], and SimII [44], hierarchically embed uni-model approaches within each other. As illustrated in Figure 2.3, each embedded approach augments the definition of one or more subsystems of the original approach. Successive refinements produce a modeling hierarchy. Each level of the hierarchy describes the operations of the model at a different level of abstraction. For developers, the hierarchy permits them to analyze the system's behavior from several perspectives and to manage effectively the growth of a complex system.

Two types of refinement methods occur in multimodel modeling. When an embedded approach matches the design of the approach it refines, the refinement is referred to as
homogeneous. For example, subsystem “T” of Figure 2.3 uses the same functional approach to simulation modeling as does the system governing its actions. The subsystem improves the definition of the system with homogeneous refinement. By contrast, when a subsystem models differently than its governing system, as illustrated by subsystems “Q” and “S” of Figure 2.3, the refinement is called heterogeneous. The subsystem improves the definition of the system with an alternative method.

2.2.3 Complexity Management

Like graphics tools, many tools for computer simulation, especially those based upon scenario languages [130], employ software composition to improve the design and reuse of simulations. They decompose systems into numerous interacting structures, many of which are identified as either processes, blocks, or activities. Communications among the structures are inferred from their organization, such as in dataflow architectures, or decided upon by explicit means, as what occurs with function calls or callbacks. Simulation results from the direct execution of the structures by active transactions, or from the evaluation of the structures by an interpreter.

Like the tools of computer graphics, the tools of simulation using software composition suffer several drawbacks. First, the roles and goals of components vary from one tool to the next. Many tools, such as GPSS [133] and SLAMII [122], primarily support only one of the many methods for simulation modeling. To produce a multimodel simulation from these tools requires extra effort and time. Few composition techniques support facilities to integrate additional methods. They expect users to develop their simulation from a single level of abstraction. Second, the relations among compositional units usually remain fixed. Topological changes that vary the structure of a simulation over time are difficult.
or impossible to produce. Third, the building of programs as networks of compositional
units is useful to model sequence, not hierarchy. The refinement of a program involves the
replacement of a short sequence with another sequence of greater length and complexity. For
some situations, this replacement scheme obfuscates the design of a program and complicates
its re-verification.

2.2.4 Software Reuse

In simulation, the reuse of software is an important issue, but there is no specific agenda of
reuse. Reuse is commonly realized through the reuse of building blocks, and occasionally,
through the reuse of composition units that operate in similar contexts. The tools rarely
provide mechanisms or methods effectively to reuse abstractions of greater size, such as
those custom-built by developers, or to reuse the abstractions in varying contexts. The tools
aim primarily to support short-term concerns, such as rapid prototyping. However, there is
a growing trend in the field to provide such mechanisms. For example, simulation frame­
works [119, 31], composition techniques [45], and experimental frames [170, 146] introduce
methods to describe simulations as the interplay of environments and interacting compo­
nents, many of which are potentially reusable and many of which are useful for software
integration.

Similar to graphics tools, few simulation tools fully support a separable specification
that reuses easily. “When” mechanisms are separate, but “what” and “how” mechanisms
heavily intertwine. This approach permits the tools to reconfigure dynamically the repre­
sentation of a system, but not without difficulty. To separate the “what” from the “when,”
numerous tools, such as PRISM [156], must employ many logical conditions. The conditions
produce code with many breakpoints that, as McCabe’s complexity measure [97] indicates,
are cumbersome to read and difficult to reuse.6

2.3 Software Communications .

Except for the tools mentioned later in section 12.2, most computer graphics tools and
computer simulation tools provide little assistance in designing and building the communi­
cations that occur amongst the parts of a system. Most tools consist of components without
communications, components with fixed communications, or components with regulated
communications. In using the tools that provide only components, such as OBLIQ-3D [104],

6A breakpoint is a branching statement that increases the number of execution paths through a
program.
developers establish and manage all forms of communications with general programming structures. They use either implicit [54] or explicit invocation, or data encapsulation to control communications among components. These types of tools permit great flexibility, but provide limited assistance in managing the growth of complexity or encouraging the reuse of software. They suffer drawbacks similar to those that discourage the widespread use of component libraries. For instance, two components of a library coupled tightly are usually difficult to apply separately and to use in alternative contexts.

In using the tools that provide components with fixed communications, such as GROOP [84], developers either parameterize or organize the use of components. Methods to directly influence or change communications are not normally present. These types of tools are useful for rapid prototyping, but not for building programs of wide variety. The fixed communications limit the kinds of programs that may be produced. Introducing alternate forms of communications, such as dynamic communications, requires great effort.

In using the tools that provide components with regulated communications, such as TBAG [42], developers structure the organization of components. They organize components to signify the occurrence of state changes and the timing of interesting events. Normally, an evaluator interprets the components and decides how and when the communications occur. These types of tools execute more slowly, but provide the greatest flexibility.

Without adequate facilities to control communications, tools of graphics and simulation provide only partial benefit to the building of a complete program. As the software engineering community has acknowledged for many years [19], components are simply inadequate to ensure the effective building and reuse of software programs. Similar sentiments have also been expressed by members of the coordination language community [27] and the community of object-oriented design [37]. Employing only components, the tools for visual simulation distribute the code for coordinating communications throughout the body of a program. Hence, the code is neither localized nor easily changed. With this drawback, the specification of topological changes to the system of a program is difficult or nearly impossible. Topological change is important to the building of programs with many changing communications, as found in many visual simulations of moderate complexity.
Chapter 3

Illustrative Examples

This chapter and the five chapters that follow present the design of IBS and a description of its parts. This chapter introduces an illustrative example, which appears in the five successive chapters, that clarifies the presentation of IBS and demonstrates its application. The illustrative example consists of several scenes involving the movements of an articulated robot. This chapter also briefly describes the RASP TOOLKIT, a software package providing an implementation of IBS with a stylized version of C++. Many of the code segments of this thesis apply the abstractions of the RASP toolkit to describe concepts and to present a working implementation of the illustrative example.

3.1 Descriptions

The illustrative example consists of three scenes: a general example, an extended example, and a contextual example. The general example involves the movements of an articulated robot searching to activate a stationary light. Upon reaching the light, the robot selects a button that toggles the light’s state. The extended example augments the general example to introduce movements to the light, while the contextual example places the robot in a larger environment that reacts to the actions of the robot. The three scenes appear in successive chapters to help describe the design and application of IBS. The general example presents elementary concepts, while the other examples establish a larger context for presenting the concepts and themes of computer simulation employed by IBS.

The description of the three scenes, which appears in this chapter, provides an overview of the illustrative example. The overview describes the goals of each scene and presents the software networks comprising each scene’s implementation. Finer details, such as the nature of the components that comprise the software networks and the coding of the
scene's software interactions, appear in successive chapters as greater information on the design and application of IBS emerge.

### 3.1.1 General Example

The diagram in Figure 3.1 identifies the software network of a simple animation in which an articulated robot selects a button to turn on a flashing light. The system consists of six components: two controllers (HAND-CONTROLLER and POSITIONAL-CONTROLLER), three geometries (ROBOT, BUTTON, and LIGHT), and one data plotter (GRAPHER). POSITIONAL-CONTROLLER updates the world position of the robot's body while HAND-CONTROLLER updates the location of the robot's hand. GRAPHER charts the position of the robot's XY-coordinate over time. The directed arrows identify the relationships among components that the software interactions control. The solid rays identify the passing of information. One component acts as the source or controller of another by providing useful information. The stippled arrow identifies an implicit relation. It indicates that one component depends on another's state, not necessarily on the value that the other produces. For example, the light flashes by the activation of the button, not by the passing of information from the button to the light.

![Diagram of a software network of a simple animation](image)

**Figure 3.1:** A software network of a simple animation

### 3.1.2 Extended Example

The diagram in Figure 3.2 extends the original software network of Figure 3.1 with four additional software components, as indicated by the gray boxes, and with seven new relationships, as indicated by the directed arrows of greater thickness. The additional components and relationships enable the original robot to react to the changes in the light's state, as produced by the PosCONTROL, ORIENTCONTROL, and COLORCONTROL. The three controllers update the position, orientation, and color of the light, respectively. The first two
controllers operate at intervals of twenty units of time with ten units of time spaced between them. Thus, successive changes in position and orientation alternate every ten units of time. When the orientation or the position of the light changes, the illumination state of the light remains fixed. Therefore, the button, which toggles the light, produces no changes until the light halts and remains stationary. The third controller, COLORCONTROL, updates the light’s color to modify gradually the scene’s illumination.

Figure 3.2: The software network of the extended example

The extended example also introduces ROBOTB, a secondary robot. When the button attains success in changing the light’s state, the button induces either COLORCONTROL to change the red channel of the light’s color\(^1\) or POSITIONAL-CONTROLLER to advance the original robot’s movements such that the original robot keeps moving towards ROBOTB. When the original robot moves, it moves until it meets ROBOTB or until sixty units of time pass. When either event occurs, POSITIONAL-CONTROLLER restarts and continues moving the original robot towards the button. The entire process repeats until the light switches state ten times.

3.1.3 Contextual Example

The diagram in Figure 3.3 illustrates a larger scene applying the extended example of Figure 3.2. The scene contains two robots, a light, a large crane on a dock, a barge, and several boxes. The movements of the first robot delimit the actions of the other entities. When the robot stops toggling the light, the scene halts and the entire animation stops. When the robot is working and the light is on, the crane attempts to load the barge with the boxes. Loading is successful if both the barge and any box are accessible to the crane. The barge moves to and from the dock periodically. It starts its motions shortly after the two

\(^1\)It is common in graphics to identify an individual color as an intermixing of three primary colors, each of which is called a channel. One of the most popular representation of color is RGB, which applies red, green, and blue as its three channels.
robots begin to move. The second robot, ROBOTB, makes a box accessible to the crane by repetitively placing a box near the crane's base. The robot places a box by picking it up and moving it to an appropriate place. Unlike the first robot, ROBOTB suffers from periodic breakdowns. When this occurs, ROBOTB must wait for time to pass before it resumes its normal activity. The final movement of the barge brings an end to the movements of the first robot. The first robot stops as the barge comes to a halt.

Figure 3.3: An illustration of the contextual example

3.2 The RASP Toolkit

The testing and development of IBS is made possible with the RASP TOOLKIT, an experimental software package for visual simulation. RASP supports an interaction-based style of programming for animating virtual scenes. When using RASP, developers write components and organize software interactions with a stylized form of C++. The stylized code is then conjoined with a run-time evaluator, also written in C++, that interprets and executes software interactions. A complete program is tested and compiled with a standard C++ compiler. The result of compilation is an executable application with software interactions governing the communications and topology of the application according to the advancement of simulation time.

3.2.1 Application Development

As illustrated in Figure 3.4, the development of an application with RASP occurs in four steps. In step one, a developer establishes the visual look of the simulation. The developer details the scene with shapes, colors, and lights. A renderer is chosen to synthesize images and an output device is selected either to store or to display the images for viewing. In step two, the developer considers the design of the software network that defines the application.
The developer identifies the proper components and determines their interactions. If a component or an interaction is unavailable within RASP, the developer produces new code in accordance to the principles of IBP. The design of new components involves careful attention to the integration of time and to the fostering of software reuse (see chapter 8). In step three, the developer produces the scripts of the application. The developer organizes the software interactions of the software network into orderly arrangements. In step four, the developer produces the meta-scripts of the application. This involves the binding of scripts with temporal structures, such as a time graph, and the binding of the temporal structures with the toolkit's run-time evaluator.

Upon completion of the four steps, the developer iteratively refines the design of a RASP-based application by adjusting its parameters or re-implementing any of the four steps.

### 3.2.2 Software Library

The current version of the RASP TOOLKIT is 2.0. Its implementation consists of 73,500 lines of stylized C++ code partitioned into 360 classes, of which 35 represent software interactions, 100 introduce components for visualization, such as geometries and cameras, and 65 introduce abstractions for animating state, such as ports (section 4.2.2) and time graphs (section 7.3). The remainder of the classes introduce basic utilities and elementary data structures, such as arrays and vectors, common to all visual simulations.
Chapter 4

Interaction-Based Programming

It has long been recognized that the kinds of mechanisms available to support communication between software components affects the reusability of components [165, 60, 121]. Most common programming approaches couple communication mechanisms directly to components, thereby complicating the development and reusability of the components. Building reusable components within those environments requires significant effort. Purtilo, for instance, states that “[the programmer must] anticipate the program unit context of use: system calls must be installed for communicating with other program units, parameters must be marshaled for transmission, data representations may need to be coerced, and so on . . .” [125, p. 152].

Interaction-based programming (IBP) is an approach intended to improve the reusability of software components by encouraging a separation between computation and coordination. It relies upon first-class language constructs, software interactions, to moderate communications between the interfaces of software components. Software interactions permit the specification of relationships among components that are typically spread across multiple statements of multiple components when using common programming language constructs, such as function invocations. Software interactions localize and consolidate the statements that determine when and how communications occur. The collective operations of software interactions form a network of communications, as shown in Figure 4.1. With software components as nodes and software interactions as links, the “software network” simulates a dataflow network. Data flows between the software components that the software interactions join. The term topology is used to refer to a particular configuration of a software network. With IBP, the topology of the network may change as the program executes.

This chapter presents IBP and exemplifies its use in developing the general example. The chapter begins with an overview of IBP that entails a discussion of program develop-
ment and of programming benefits. With IBP, program development involves the building of programs as two parts, one part for computation and another part for coordination. Following the overview, this chapter provides a detailed description of the two parts of an IBP program and presents an application for their use. The part for computation involves the development and use of software interactions and software components, and the part for coordination involves the development and use of scripts and meta-scripts. The chapter concludes with a discussion of program execution. It explains how a special evaluator interprets the two parts of an IBP program to establish a working system that may change its topology at run-time.

4.1 Overview

An interaction-based program organizes software components with software interactions. The program relies upon this organization to form systems of communicating components. The main body of an interaction-based program is unlike that of programs written with conventional programming languages. As discussed in section 4.4.1, its execution does not produce immediate results or initiate communications. The execution of an interaction-based program imbues software interactions with argument values and uses them to form descriptions of software networks. Communications occur when the descriptions are interpreted, not when the descriptions are formed. Thus, an interaction-based program shares greater commonalities with the script of a theatrical production. It describes the behaviors and interactions of its constituents by indicating for each scene who participates, what happens, and when the scene occurs.
4.1.1 Program Development

The focus of an interaction-based program resembles that intended by the designers of many scripting languages, such as TCL [110] and PERL [160]. As noted by Ousterhout, the developer of Tcl, scripting languages simplify the production of connections between components and provide the means to build applications quickly [111]. General coding techniques offer the best means to define the data structures and algorithms of software components. Employing these techniques, the development of an interaction-based program involves the building of two activities, activities for production and activities for coordination. The former decides the computational state of a system while the latter decides the communications of the system that regulate the computational state. The activities for production establish the computational state by transforming inputs to outputs, and by communicating information between the outputs and the inputs of individual components. The activities for coordination establish the communications of a system by structuring the use of the activities for production. Collectively, the two activities naturally partition a program into parts for computation and parts for communication. Unlike a program developed with common software practices, such as procedural or object-oriented programming, an interaction-based program enables its computation and communications to be produced and updated individually. The code for each part is localized and isolated from the other.

Activities for Production

The development of the activities for production involves the design and implementation of software components and software interactions. Software components maintain and update state information according to a set of changing input values. As presented in the example of section 3.1.1, software components receive messages as inputs. The messages emanate from software interactions while communicating information from the components’ environment or from the components’ peers. Software interactions send messages to software components by referencing the software components by their operations. Upon invoking the operations, the software interactions call a variety of actions either to update or to alter the communication of information, such as filtering the information to reduce its value or limiting the information to pass only when specific run-time conditions apply.

Activities for Coordination

The development of the activities for coordination involves the design of scripts and metascripts, two independent abstractions that are assembled from an organization of parts.
Scripts detail the use of software interactions in coordinating the communications between a variety of software components; each software interaction assumes a specific role in building and modifying the computational state of a program. Each component receives a participatory role in contributing to the development of a software network of fixed topology. When the script executes, the components act out their roles and maintain continual relations with their peers.

Meta-scripts define a larger blueprint for the collaborative workings of numerous scripts. Each script receives a participatory role in introducing a new software network to a program's execution. As individual scripts execute, the topology of the program's network of communications changes. Unlike that which occurs with traditional programming methods, such as object-oriented programming, the communications between software components is not realized until the program executes and applies its software interactions dynamically.  

4.1.2 Programming Benefits

As a program becomes large and complex, the separation between the two activities gives rise to several programming benefits. First, the separation facilitates the rapid identification and manipulation of the coding that controls when and why components communicate. The coding for controlling state remains local to the definition of the activities for production. The coding does not intermix with the coding for controlling communications. Hence, there is less clutter to dissect and to extract when installing changes to activities of either type. Second, the separation supports popular component technologies, such as CORBA [107], which apply a building block approach to developing application programs. Components are fundamental abstractions that IBP uses to produce the activities for production. IBP requires only that components adhere to the tenets of IBP, as discussed in chapter 8.

4.2 Activities for Production

The activities for production consist of software components and software interactions. Software components compute the state of a system while software interactions configure the communications of the system.

\[\text{\footnotesize \textsuperscript{1}}\] The activities for coordination assemble and structure programs similarly to several concurrency models, such as ACTOR-based systems [2], CSP [65], and $\pi$-CALCULUS [102]. Each of the concurrency models apply either high-order abstractions or special notations to express or monitor patterns of communications between collections of components (or processes).
4.2.1 Software Components

In IBP, software components are elemental abstractions that partition a system into a collection of interacting parts. Similar to those of component-based systems, such as MILs (section 12.1.2) and ADLs (section 12.1.3), these elemental abstractions determine a program's design and configuration. Software components are the primary entities of a program that establish state and exhibit behavior. IBP requires that software components send and receive information via ports, first-class operations for defining a component's interface. The ports permit software interactions to reference and to invoke a software component by its operations. A software interaction explicitly manipulates a port to send and accept information when required. Conceptually similar to the ports of software architectures [9, 52], the ports of IBP establish a logical interaction between a component and its environment.

The illustration in Figure 4.2 presents a component with an interface consisting of five unidirectional ports (see section 4.2.2). The ports permit information to flow in and out of the component in only one direction. The code at the right of Figure 4.2 exemplifies the usage of the component by its ports. The variables begin and val reference respectively two individuals ports, inBegin and outVal (lines 2-3). Normally done within the body of a software interaction, the two variables invoke functions on the ports either to communicate information or to query a port's state. The variable begin invokes setValue to communicate the value 10 to the component (line 4) while the variable val invokes getValue and getType to obtain a value and determine what type of information the port provides (line 5).

```c
class Controller {  
  public:  
    Controller hand;  
    Inport *begin = hand->inBegin();  
    Outport *val = hand->outVal();  
    begin->setValue(10);  
    cout << val->getValue() << val->getType() << endl;  
};
```

Figure 4.2: A component with five unidirectional ports

4.2.2 Software Interactions

A software interaction is a first-class mechanism that forms a basic, indirect relation between software components. It communicates information among components to incite collaboration and to initiate the sharing of data. A software interaction permits components to communicate anonymously. Components neither maintain references to their peers nor inform their peers to signal the occurrence of important events. All communications among
components occur indirectly via the execution of a software interaction. The design and use of a software interaction is unlike those of general mechanisms for communication, such as implicit or explicit invocation [108]. A software interaction separates the specification and the execution of commands that form communications. A software interaction remains inactive until it receives a notice to communicate, which may arrive from a variety of sources. As explored in this thesis, a notice to communicate may originate from high-level abstractions that relate the execution of software interactions to the advancement of time or to the occurrence of interesting events.

For instance, the code in Table 4.1 presents two separate code segments for communicating information between two components, a and b. The first segment applies a software interaction between the components, while the second segment has the components communicate directly. The first segment initializes the software interaction si with an invocation to si.set(). The command provides si with the identity of the two ports, one port of component a and one port of component b. That command is neither responsible nor capable of inciting the actual communications. Communications occur when the command si.execute() instructs the software interaction to communicate. Each time the command is invoked, the command passes information between the two ports. In contrast, the second segment applies an explicit invocation to communicate information directly from a to b. Embedded within the definition of component a, the communications between the two components is direct and immediate. Component a recognizes the identity of b and the invocation occurs immediately after it is specified. Repetitive calls to communicate, as indicated by the two calls to b->in(10), require that the receiver of information be repetitively identified.

<table>
<thead>
<tr>
<th>Software Interaction</th>
<th>Explicit Invocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interaction si;</td>
<td>define component a()</td>
</tr>
<tr>
<td>si.set( a-&gt;out(), b-&gt;in() );</td>
<td>{</td>
</tr>
<tr>
<td>si.execute();</td>
<td>b-&gt;in( 10 );</td>
</tr>
<tr>
<td>si.execute();</td>
<td>b-&gt;in( 10 );</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

Table 4.1: Two means of passing information from a and b

By separating the specification of communications from its execution, a software interaction delays before operating. It may communicate information long after it is initially applied. For visual simulation, delayed communications are critical. Not all communication in a dynamic system occurs immediately. Most communications remain inactive until important times or states arise. Throughout the period of a program's execution, a software
CHAPTER 4. INTERACTION-BASED PROGRAMMING

interaction may repetitively toggle between an active and inactive state. The toggling of a software interaction’s state produces a program with dynamically-changing communications.

Operations

A software interaction operates by accessing the ports of components. It invokes the methods of the ports to decide whether the ports are ready and able to communicate information properly. Table 4.2 presents three common queries performed by a software interaction on a port. A DATA TYPE query identifies the data types that a port accepts or returns. A software interaction ascertains that information to ensure that the components that it interconnects are compatible. A RELATION query identifies the ports of components that exhibit similar behaviors, such as those that update the same state variables. Two ports that behave similarly should probably not be invoked at the same time. A PAST ACTION query identifies the last instance when a port has communicated information properly. In a time-varying system, one port should not be permitted to accept two values at the same moment in simulation time.

<table>
<thead>
<tr>
<th>Query</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA TYPE</td>
<td>what type of data type does the port accept or return?</td>
</tr>
<tr>
<td>RELATION</td>
<td>with what other port does the operation behave similarly?</td>
</tr>
<tr>
<td>PAST ACTION</td>
<td>at what time did the port operate last</td>
</tr>
</tbody>
</table>

Table 4.2: Common query types of first-class operations

For a software interaction to operate properly, it is preferable that ports be either unidirectional read or unidirectional write. Both kinds of ports ensure only one-way flow of information to or from a component.\(^2\) It is the software interaction, not the port, that ultimately determines when and how information flows. A unidirectional read port provides information to a software interaction without requiring any input parameters: the port need not consume input in order to supply output. A unidirectional write port behaves similarly with information flowing in the opposite direction. In this case, a software interaction passes information to the operation without the expectation that the port will return any outgoing results.

A software interaction does not readily accept the use of bi-directional ports. Those kinds of ports hinder a software interaction from operating freely. Bi-directional ports require

\(^2\)With the aid of a software interaction, the binding of an outport and an inport from the same component produces a unidirectional cycle. The cycle is not a producer of bi-directional flow because information always travels in and out of the component in one direction.
that information flow in and out (or vice-versa) of a component in a fixed sequence. A software interaction is not freely permitted to control each flow individually. Hence, such a port undermines the authority of the interaction to choose when and how communications occur. For example, the bi-directional read port requires an argument value before it returns a result. A software interaction is not permitted simply to read the result without writing an argument. The design of the bi-directional read port also precludes software interactions from observing the results of previously written arguments. To achieve the proper results, the software interaction must either supply the port with duplicate arguments or inform the port to return its most recent result. Either way, each approach introduces additional complications. The former approach requires an interaction to share argument values while the latter approach requires an interaction to identify itself as a “primary” or “secondary” reader. A primary reader requests new information while a secondary reader queries for information given to previous requests.

In some circumstances, a software interaction benefits from the use of a bi-directional write port. The port returns a result “before” it accepts information. Unlike a function call, it supplies the caller with argument values. For a software interaction, those operations help to perform a partial write, the writing of incomplete information. (see chapter 8 for further details).

Attributes

There are many kinds of software interactions. Most are small in code size and produce a specific effect that is easy to comprehend and relatively easy to adapt. Some software interactions are domain-specific while others apply equally well to several domains. The ones that are presented in this work are well-suited for visual simulation and other dynamic systems. They observe the passage of time and produce time-varying communications. These software interactions are characterized by seven attributes, which appear in Table 4.3. Software interactions with comparable attributes are of the same kind and facilitate communications similarly.

The ACTION attribute expresses the task of a software interaction and identifies the method by which the software interaction achieves its task. For example, a simple software interaction passes information between components with standard function calls. An interaction of greater complexity may employ a remote procedure call to move the information among components that reside on a distributed network, or an implicit invocation to register callbacks from one component to another.

The PORT attribute identifies the number and kinds of first-class operations that
CHAPTER 4. INTERACTION-BASED PROGRAMMING

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTION</td>
<td>specifies the task of the interaction and the method for achieving the task</td>
</tr>
<tr>
<td>PORT</td>
<td>identifies the number and kinds of ports used by the interaction</td>
</tr>
<tr>
<td>MODE</td>
<td>identifies the partition of ports according to their arrangement and usage</td>
</tr>
<tr>
<td>FILTER</td>
<td>specifies the kind of filter, if any, used by the interaction</td>
</tr>
<tr>
<td>CONDITION</td>
<td>identifies the conditions that affect the interaction's operations</td>
</tr>
<tr>
<td>TEMPORAL</td>
<td>specifies the temporal aspects, if any, of the interaction</td>
</tr>
<tr>
<td>FAILURE</td>
<td>indicates how the interaction responds to failure</td>
</tr>
<tr>
<td>PARALLELISM</td>
<td>indicates if an interaction performs any of its actions in parallel</td>
</tr>
</tbody>
</table>

Table 4.3: Attributes of software interactions

A software interaction accepts and the types of information that the software interaction requires from its operations. Most often, this attribute is divided into "inports" and "outports." The former indicates (unidirectional) read operations while the latter refers to (unidirectional) write operations. Unless an appropriate filter is available, an operation that does not satisfy the data type requirement of the software interaction is identified as unacceptable or improper.

The MODE attribute expresses an arrangement and use of ports by a software interaction. An arrangement determines how ports are applied as a software interaction operates. It is not uncommon for a software interaction that accepts multiple ports to partition its ports into several modes, of which only one is usually active. The active mode is referred to as the operating mode.

The FILTER attribute indicates whether a software interaction accepts a filter and, if so, what type of filter it accepts. A filter is useful for transforming or updating information that does not transfer easily between normally incompatible ports. For example, information does not flow properly between an outport producing a three-dimensional value \((x,y,z)\) and an inport accepting only a single dimensional value. A filter enabling a software interaction to transfer information properly between the two ports would simplify a three-dimensional value into a single value using a decimation technique, such as selection or reduction [67].

The CONDITION attribute indicates whether a software interaction works under any conditions, and if so, it identifies the conditions. Normally, a condition influences the behavior of a software interaction by altering how the software interaction mediates communications. A condition may either halt or alter the mode of the execution of the software interaction. Two common conditions that affect a software interaction are "time" and "state." The former identifies time as a regulatory factor. Under that condition, a software interaction operates according to the passage of time or the advent of future moments. The latter identifies state information as a determining factor. Under that condition, a software interaction
operates according to the current or upcoming value of state information, which is usually obtained from one or more inports.

The TEMPORAL attribute indicates whether a software interaction communicates temporal information, and if so, indicates what the temporal information represents. Normally, a software interaction uses temporal information to synchronize the behaviors of software components operating in time-varying environments. For instance, a software interaction may communicate a duration of time to a port for which a software component bases its computations.

The FAILURE attribute indicates whether a software interaction reacts to a failure, and if so, identifies how the software interaction reacts. Normally, a failure arises if a software interaction is unable to execute properly, to access its ports, or to satisfy its conditions. For most software interactions, failure is met by either doing nothing, waiting for success, or signaling an error. A software interaction doing nothing remains idle. A software interaction waiting for success hopes for the failure to be resolved. While the software interaction waits, it usually halts the execution of successive interactions until the failure disappears. A software interaction signaling an error indicates a major problem. It reacts by halting the evaluation of additional software interactions or by terminating the execution of the entire program.

The PARALLELISM attribute indicates whether a software interaction performs any of its actions in parallel, and if so, it identifies those actions. For instance, a software interaction working with multiple ports may communicate information between the ports concurrently, if its underlying hardware permits.

4.2.3 General Example Revisited

The diagram in Figure 4.3 adds details to the original diagram of Figure 3.1. It presents software components with unidirectional ports, and communications with software interactions. The unidirectional ports permit the software interactions to control freely the flow of information to and from a component. For example, the ports inHand and outHand of the ROBOT component permit a software interaction to read or to write separately the position of the robot's hand. The software interaction D provides information directly to inHand without receiving any results. The software interactions A and F freely determine when and where to send the results of the positional update. They operate properly without knowing which software interactions initiated the update or which components provided the ROBOT with an updated value.

The attributes of the software interactions found in Figure 4.3 appear in Table 4.4.
The most commonly applied software interaction, which forwards information from an outport to an inport, is of type \textbf{Event}. In the diagram, this kind of interaction appears eight times (A-F, P-Q). The software interaction F differs slightly from the others in that it employs a special filter to extract a two-dimensional value from \texttt{outHand}, an outport that identifies the position of the robot’s hand in three dimensions.

![Diagram](image)

**Figure 4.3:** The general example revisited

<table>
<thead>
<tr>
<th>Attribute</th>
<th>A-F,P-Q</th>
<th>I</th>
<th>Y</th>
<th>X</th>
<th>M</th>
<th>M’</th>
</tr>
</thead>
<tbody>
<tr>
<td>PORT</td>
<td>1 outport &amp; 1 inport</td>
<td>1 outport &amp; multiple inports</td>
<td>multiple inports</td>
<td>multiple inports</td>
<td>1 “bool” output</td>
<td>multiple “void” inports</td>
</tr>
<tr>
<td>ACTION</td>
<td>pass info from outport to inport</td>
<td>pass info from outport to multiple inports</td>
<td>pass current timestep to inport</td>
<td>pass current duration to inport</td>
<td>invokes other events according to value of output</td>
<td>invokes an input</td>
</tr>
<tr>
<td>PARALLELISM</td>
<td>NA</td>
<td>passing info to imports</td>
<td>passing time to imports</td>
<td>passing info to imports</td>
<td>NA</td>
<td>calling imports</td>
</tr>
<tr>
<td>FILTER</td>
<td>[transforms info]</td>
<td>[transform info]</td>
<td>[transform info]</td>
<td>[transform info]</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>CONDITION</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>waits for true value</td>
<td>NA</td>
</tr>
<tr>
<td>TEMPORAL</td>
<td>NA</td>
<td>NA</td>
<td>timestep</td>
<td>duration</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>FAILURE</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MODE</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>true</td>
<td></td>
</tr>
</tbody>
</table>

**Table 4.4:** Attributes of the software interactions in Figure 4.3

The software interaction I is a \textbf{FanEvent}, a specialized interaction that accepts multiple inports. Upon activation, the software interaction passes information concurrently from a single outport to each of its inports. The software interactions X and Y provide the
HandController and the PositionalController with temporal information. \( X \), of type \( \text{TimeEvent} \), communicates the advancement of time while \( Y \), of type \( \text{DurationEvent} \), communicates a duration of time for the two controllers to apply to their computations.

The software interaction \( M \), of type \( \text{CondEvent} \), is a conditional event. That software interaction accepts one outport and two sets of interactions. Upon activation, \( M \) invokes one of the two sets of interactions according to the logical value of the outport. When the outport is true, \( M \) invokes \( M' \) to invoke the toggle port of the light. \( M' \), of type \( \text{CallEvent} \), is a software interaction that invokes a "void" inport, an operation requiring no input values. When provided with information, the inport sends a dataless signal to its component.

As discussed in section 3.1.1, the goal of the software network in Figure 4.3 is to coerce a robotic device to toggle a light source. The software network accomplishes its goal by communicating information among eight software components. Interactions \( C \) and \( D \) establish links between the two controllers and the robot. The robot's orientation and position are set by the arrival of information from the two controllers, HandController and PositionalController. Interactions \( A \) and \( B \) parameterize the controllers with initial values. Each controller is set to start with an initial value that originates from the robot. Interactions \( X \) and \( Y \) disseminate temporal information to the two controllers. \( X \) provides a durational value while \( Y \) provides a value of temporal advancement. The repeated execution of \( Y \) decides how often the controllers update to the advancement of time. Finally, interactions \( M \) and \( F \) operate to toggle the light and provide the Grapher with coordinate values.

The components Distancer and Subtractor are secondary software components that perform numerical operations on positional data. Interactions \( E \) and \( I \) provide Distancer with two positions for computing a distance while \( P \) and \( I \) provide Subtractor with two positions for computing an offset. Interaction \( Q \) communicates the result of Subtractor to HandController. Both Distancer and Subtractor serve only to adjust the values communicated between components. As will be discussed in chapter 5, these components are unnecessary when applying interaction constraints to the network of connections.

### 4.3 Activities for Coordination

The activities for coordination consist of scripts and meta-scripts. Scripts organize software interactions while meta-scripts control scripts (and other meta-scripts).
4.3.1 Scripts

A script is an orderly arrangement of software interactions that decides how numerous software interactions collaborate to produce a software network, a system of communicating components. The arrangement of software interactions within a script determines how a software network forms and what topology the software network assumes. The software network forms when a run-time evaluator (RTE) interprets the script and controls the execution of software interactions (see section 4.4.1). As the RTE operates, it toggles the script between two states, active and inactive. While active, the script produces communications. As the script toggles between the two states, the script alters the topology of the program so as to replace old communications with new ones.

A script assigns meaning to the sequence of communications. Unlike a procedure, a script excludes commands that explicitly define and establish a computational state. A script isolates the communications of a system from commands that control state. In a procedure, the intent and meaning of communications are presented as a collection of uncorrelated messages or function calls. The communicational statements of the procedure are cluttered with commands for computing state values. As noted by various academics [22], a sequence of uncorrelated method invocations is difficult to decipher and reuse when the complexity of a system grows. A general mechanism, such as a script, is critical to the effective reuse and understanding of the communications that occur between collaborating components.

Arrangements

The means to organize and control the arrangement of software interactions in a script varies widely. Each means presents an alternative set of criteria for deciding when and how software interactions operate. The criteria apply to software interactions on their own or as a part of a larger set. Various criteria that are useful, especially to visual simulation, appear in Table 4.5.

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEQUENCE</td>
<td>orders software interactions</td>
</tr>
<tr>
<td>CONCURRENCY</td>
<td>executes software interactions simultaneously</td>
</tr>
<tr>
<td>DEPENDENCY</td>
<td>relates the execution of software interactions</td>
</tr>
<tr>
<td>FREQUENCY</td>
<td>establishes a pattern of repetition for software interactions</td>
</tr>
</tbody>
</table>

Table 4.5: Attributes of software interactions

The SEQUENCE criterion identifies an ordering of software interactions. Once begun, the software interactions activate sequentially. The successive activation of several software
interactions are useful for controlling the flow of information through a network of coupled components. Each software interaction moves information from one component to the next.

The *concurrency* criterion expresses the simultaneous execution of multiple software interactions. As the software interactions execute, they regulate communications concurrently. For many dynamic systems, concurrency is an important concern that affects performance and decides the results of analysis.

The *dependency* criterion identifies all relations among software interactions that influence their activation. One interaction may depend upon the activation or deactivation of others before it operates. The specification of relations are useful for describing systems with many software interactions that rely upon the past (or future) activations of others.

The *frequency* criterion identifies a pattern of repetition for a software interaction. It indicates how often it repeats and the number of times it occurs. In many systems, such as those that model continuous state, repetitive communications occur frequently. The frequency with which components communicate often decides the precision of state changes. Many numerical methods \[120\] are sensitive to the frequency of their execution.

### 4.3.2 Meta-Scripts

A meta-script establishes a plan for the execution of one or more scripts. It overlays the scripts with information that decides when and how the scripts of a program toggle on and off, and occasionally, when and how multiple meta-scripts execute. The information applied by a meta-script is specific to the system being produced. For visual simulation, a meta-script associates scripts with activation times and with temporal relations. The activation times determine how scripts execute over time. As simulation time progresses, scripts with early activation times execute first. Temporal relations associate the activation time of scripts to the activation times of other scripts. For instance, one script may begin after the termination of another.

A meta-script is similar to a script in that each applies similar organizational criteria (see section 4.3.1). The organizational criteria of a script, such as sequence and concurrency, apply equally well to deciding the execution of meta-scripts. However, a meta-script differs from a script in that a meta-script applies its criteria to an “interval of execution.” Unlike a software interaction, a script is a continuous action with a beginning and an end.\(^3\) Therefore, the criteria for organizing a script applies equally well to its beginning and to its end, and occasionally, to both. For example, the sequencing of two successive scripts by

---

\(^3\)When the beginning and end of a script coincide, the script acts instantaneously.
their beginnings does not naturally produce a simple linear sequence. If the second script terminates early, the first script will finish last.

4.3.3 General Example Revisited

The code in Figure 4.4 presents an implementation of the general example of Figure 3.1. Using the RASP toolkit, the code consists of two parts, one part declaring production activities and another part producing a script, entitled LightByRobot. The declaration of production activities defines the components and the software interactions of the script. The script parameterizes the software interaction with ports, and chooses a plan for executing the software interactions over time. With IBP, the coding for two parts of the program appears separately. The definition of the script does not depend on the design or identity of the production activities. The script operates equally well with other software interactions or with other software components providing similar ports. The script simply organizes the definition and application of the software interactions.

The script LightByRobot organizes software interactions with an activity, entitled act (lines 16-18). The activity partitions the software interactions into two sequences that begin at different times and execute with different frequencies. The first sequence activates its software interactions immediately after the script begins (line 16). The activation of those software interactions occurs only once. The sequence begins with evfI and ends with evA + evB. The + operator indicates the parallel activation of software interactions. The software interaction executes in parallel if the underlying hardware permits. The second sequence executes its software interactions repeatedly while the script remains active (lines 17-18). For every time step, the sequence begins with tevY and ends with evQ.

The code in Figure 4.5 presents the meta-script for "RoboWorld," a simple animation involving a robot that sits, spins, waves, and turns on a light. The meta-script animates the robot with four scripts, of which LightByRobot is one. The meta-script associates temporal information with the scripts by way of TimingActs (lines 5-6,8-9). The TimingActs store the times when the scripts begin and end, and the rates at which the scripts advance with simulation time. The temporal instructions of the meta-script instruct the robot to turn on the light for one hundred units of time (line 9) after the robot spins around its axis for seventy-five units of time (line 6). While the robot spins about its axis, the meta-script instructs the robot to wave its hand simultaneously. The simultaneity of the scripts for each action is accomplished with a relation that delimits one script to the execution of another (line 7). Another relation, MEET, forms a dependency between lightByRobot and sit (line 10). The robot sits down after it finishes its motions to trigger the flashing light.
4.4 Program Execution

The execution of an interaction-based program differs from that of conventional programs in three ways. First, an interaction-based program executes by way of a run-time evaluator (RTE), a special module that interprets the scripts and meta-scripts of a program so as to enforce software interactions to communicate information. Second, an interaction-based program operates in three phases. The first phase creates scripts and meta-scripts while the remaining two phases apply the scripts and meta-scripts in structuring a program's
/* production activities of RoboWorld */
(1) Activity *lightByRobot = LightByRobot();
(2) Activity *spinningRobot = SpinningRobot();
(3) Activity *wavingRobot = RunningRobot();
(4) Activity *sittingRobot = SittingRobot();
/* wave and spin occur simultaneously */
(5) TimingAct *wave = new TimingAct( wavingRobot );
(6) TimingAct *spin = new TimingAct( spinningRobot, 100, 175 );
(7) spin->setRel( DELIMITS, wave );
/* sit occurs after light */
(8) TimingAct *sit = new TimingAct( sittingRobot );
(9) TimingAct *light = new TimingAct( lightByRobot, 250, 350 );
(10) light->setRel( MEETS, sit, 75 );

Figure 4.5: A meta-script for animating the movements a robot

topology and controlling a program’s state. Third, an interaction-based program experiences
changes to its topology. As scripts and meta-scripts dynamically mediate communications,
the topology of the program changes with its execution.

4.4.1 Run-Time Evaluator

The integration of the RTE with an interaction-based program produces an executable application. As shown in Figure 4.7, a standard compiler integrates the RTE with an IBP program by parsing and linking the definition of each. The compiler ensures that the program and the RTE properly follow the syntax of an appropriate host language, such as C++.

When the executable application is applied, it prepares the RTE to interpret a program by its activities for coordination. By examining the activities for coordination carefully, the RTE is capable of executing software interactions in parallel, optimizing run-time performance, and flagging run-time errors.

The design of the RTE varies according to the domain of its application, not by its
intended use. The design varies according to the methods by which a program manages its scripts. In time-varying systems, for example, the RTE manages the advancement of time. As time advances, the RTE examines the timing values of scripts and evaluates those ready for activation. Timing values, such as sequence and temporal intervals [7], are set with the meta-scripts of a program. The application of the RTE is relatively transparent to the designers of an interaction-based program. The designers simply invoke the RTE to initiate the execution of IBP programs. The RTE operates independently of the design and behavior of the software components that define the program.

4.4.2 Three Phases of Execution

As illustrated in Figure 4.8, an IBP program operates in three phases, referred to as *script execution*, *RTE execution*, and *system execution*. During script execution, an IBP program creates scripts and meta-scripts. It executes code to form, organize, and specify the arrangement and execution of software interactions. During RTE execution, the RTE interprets the scripts and meta-scripts of the first phase. The RTE executes software interactions so as to communicate information between software components. During system execution, the software components of the program communicate and compute state. It is during system execution that a program resolves its primary task: the algorithmic solution to a computational task.

The latter two phases, RTE execution and system execution, occur concurrently. As
the RTE induces communications, the software components send and receive information as they actively compute the program's state. As the RTE progressively interprets the scripts (and meta-scripts) of a program, the topology of the system changes and the components receive new communications, upon which the RTE computes new state information.

4.4.3 Dynamic Topology

As an interaction-based program executes, it produces a software network resembling a dynamic, integrated dataflow network. Information passes dynamically between software components interconnected by the software interactions of the program. The continual activation and deactivation of the scripts (and meta-scripts) of a program dynamically alters the size and topology of the network's configuration. With each new topology change, the software network assumes a new configuration and a new set of interactions. The newly activated interactions replace existing ones, and thereby establish new dependencies between software components. As the new dependencies form, the software components acquire new roles and undergo new settings. They experience an alternative form of software reuse, which occurs during the program's execution. Each of the scripts of a program may introduce many alternative uses for one component in varying contexts. As noted in chapter 8, IBP requires that software components accept and accommodate this form of execution to improve the use and understanding of IBP code.

If an RTE executes scripts in parallel, as performed with visual simulations, a software network undergoes concurrent changes to its topology. As the RTE interprets each script, it decides when and how to further advance the topology of the overall network. In a time-based system, the RTE executes every script according to the advancement of simulation time. As time flows, the RTE handles the simultaneous exchange of information amongst the many parts of a program. Whenever the RTE discovers an inconsistency in the networks, such as the simultaneous access and update of a single port, it signals an error in the program's design or execution. If a program's design is in error, its implementation is incorrect. The program is not designed well to perform its intended task. If a program's execution is in error, its ordering of scripts and meta-scripts is incorrect. The program is not designed to apply its communications properly.

Benefits

For many application domains, such as visual simulation, topological change is an important tool for the development and maintenance of complex systems. As noted by many [21, 164,
CHAPTER 4. INTERACTION-BASED PROGRAMMING

115], it permits a system to introduce new components by either updating or removing old components, and by reconfiguring the mapping of software to hardware. With IBP, the benefits of producing a program with dynamic topology are two-fold: a program can develop from a gradual process of topological change, and a program can easily accommodate new configurations. The configuration of a program derives from its execution, not from its structure as specified by its implementation.

When a program develops gradually, it adapts its topology either to match run-time conditions or to perform new tasks. Unlike conventional implementations, the program avoids having to configure itself as a single system that matches all possible conditions and performs all required tasks. A single system grows large and cumbersome as it expands to handle additional conditions or tasks, or any combination of the two. With traditional programming approaches, the coding of a single system entails the intermixing of many control statements with numerous computational commands. As the intermixing increases, the single system grows difficult to decipher and to change.

The two diagrams in Figure 4.9, for example, apply separate approaches in developing the same program. The diagram on the left portrays the program as an evolution of three networks. The evolution of the networks involves a gradual process of topological change. Each network derives from a separate set of controls. The introduction of a new condition or task to the program involves the definition of a new network and the specification of controls for managing it. The diagram on the right portrays the program as a single network. The network integrates the three networks of the first approach and their respective controls into one system. To decipher or update the system requires great effort because the controls of all three individual networks are present for producing the network as a single system. Those controls coincide with the statements that decide when and how the network evolves and the individual components communicate.

![Topological Changes](image1.jpg)  ![Complete Integration](image2.jpg)

Figure 4.9: A comparison between a program that changes topology and a program that consists of a single network

A program that accommodates new configurations is helpful in assisting developers in
CHAPTER 4. INTERACTION-BASED PROGRAMMING

understanding how a program works and what parts a program applies during its execution. While it executes, a program with dynamic topology may answer queries about its structure and about the parts that are actively engaged in communications. The program does this by examining the activity of its scripts and the attributes of its software interactions. Developers alter undesirable configurations by adding, removing, or altering scripts and meta-scripts. A program that changes its configuration is also capable of accommodating a broader variety of changes to its definition than that of a program devised as a single system. A single system grows large and complex as it accommodates new features and enhancements. Unforeseen changes cause trouble if the program is unable to accommodate the changes smoothly. A program with changing topology accepts unforeseen changes by introducing a new script or meta-script that structures the program appropriately. The unforeseen changes neither require nor cause extensive modifications to the program’s design and complexity, unlike what often occurs with a program designed as a single system.

4.4.4 General Example Revisited

The coding of the general example that appears in Figures 4.4 and 4.5 executes by an RTE designed specifically for visual simulations. The RTE executes in conjunction with the passage of simulation time. When the complete program begins, it begins with “script execution.” The program defines the scripts and meta-scripts of the program, such as LightByRobot, and prepares them for the RTE to examine. Afterwards, the program performs “RTE execution.” The RTE mediates communications, which in turn launches “system execution.” While the RTE invokes the interactions of LightByRobot, the software components of the program, such as the hand controller, compute and update the program’s state.

The program undergoes topological changes as the many scripts and meta-scripts of the program operate over time. The robot, for example, can acquire a new role with each change to the topology of the network. As with the network shown in Figure 4.3, the robot might begin by consuming information from a positioner controller. After a short delay, another script could augment the network to have the robot also provide information to another robot seeking to meet the original robot’s position.
Chapter 5

Interaction Constraints

Interaction constraints produce temporary relations between software interactions and between the ports of software components. They simplify the production of scripts by diminishing the number of ports, software interactions, and software components required to develop programs with complex communications. Interaction constraints consists of virtual ports and virtual events. Virtual ports relate the binding of ports, while virtual events relate the scheduling of software interactions. Collectively, both establish conditions, logical and numerical, to check, restrict, or compel the flow of information between two or more software components.

5.1 Foundation

Interaction constraints are founded upon one-way constraints, a popular approach for applying and evaluating unidirectional relationships [117]. With one-way constraints, the relations among ports and software interactions are represented as equations with variables, of which one variable is the primary recipient of a particular relation. The recipient variable updates state whenever any of the remaining variables, referred to as independents, change state. A one-way constraint is satisfied when its state reflects the value of its independents. For example, in the one-way constraint \( z = x + y \), \( z \) receives the value of \( x \) plus \( y \). If either \( x \) or \( y \) changes, the equation recomputes \( z \). In a one-way constraint, the resolution of a relation occurs in only one direction: the direction of the variable receiving information.

One-way constraints help simplify the expression of complex interactions as they permit the interactions to be stated declaratively. One-way constraints have been applied successfully to many forms of visual simulation, especially to the development of graphical user interfaces. In 2D graphics (see [131] for a survey), one-way constraints, such as those
of SKETCHPAD [150] and COOL [75], have been applied to geometric layout and interactive design. In 3D graphics, one-way constraints, such as those of CONDOR [76] and AVS [153], help maintain dependencies between dataflow elements and computational structures.

5.2 Virtual Ports

A virtual port is a short-hand mechanism for expressing one or more interactions among several software components. The virtual port employs operators — mathematical, functional, or logical — to express symbolic relationships among (unidirectional) ports. The operators form a one-way constraint among the ports that is satisfied each time the virtual port receives a request for information. The request may come from either a software interaction or another virtual port. A virtual port responds to a request for information by applying the operators to the ports that it relates. It applies the operators when they are required, not when the operators are first interpreted. Hence, the operators do not produce immediate results, as they normally do in standard imperative programming. A virtual port acts lazily until its value is required.

The code in Figure 5.1 presents exemplary definitions of three virtual ports: add, examine, and either (lines 1-3). Each of the virtual ports apply a single operator to the positional values of two poles. add sums the positions while examine and either compare and select the positions respectively. Each of the ports applies its operator when responding to a request. either, for instance, returns non-deterministically the position of either pole when the interaction evtl requests the port’s value (line 4).

```
examples of virtual ports
(1) VPort *add = *pole1->outPosition() + *pole2->outPosition();
(2) VPort *examine = *pole1->outPosition() < *pole2->outPosition();
(3) VPort *either = *pole1->outPosition() || *pole2->outPosition();
interaction using virtual port
(4) Event *evtl = new Event( either, ball->inPosition() );
```

Figure 5.1: Exemplary definitions of three virtual ports

5.2.1 Backwards Propagation

When a virtual port applies its operators to other virtual ports, it initiates a request for information that propagates backwards. Each of the other virtual ports acknowledges the

---

¹One position is less than another position if all its dimensions are less.
request for information by applying their own operators to their own ports, which may consist of additional references to other virtual ports. Applying the same algorithm as that of THINGLAB [23], the backwards search terminates when the value of every involved virtual port is found.\(^2\)

The short code segment in Figure 5.2 illustrates the use of virtual ports and the search that arises when one virtual port references another. The virtual ports appearing in the code segment are \texttt{middle} and \texttt{bNear} (lines 1-2). As shown in Figure 5.3, \texttt{middle} computes the center of two poles while \texttt{bNear} determines if the middle of the two poles is less than ten units away from a ball. Each time \texttt{cEvt} activates, it calls upon \texttt{bNear} to determine if the middle of the two poles is near the ball’s current position. \texttt{bNear} computes the distance between the two objects by invoking the function \texttt{dist} and requesting information from \texttt{middle}. If \texttt{cEvt} obtains a positive response from \texttt{bNear}, it invokes \texttt{evtl}, which sets the ball’s position to match the middle of the two poles. Since \texttt{middle} evaluates at run-time, it always computes correctly the midpoint of the two poles, even if the two poles are continually moving.

\begin{verbatim}
establish constraint between poles and ball
(1) VPort *middle = *(*pole1->outPosition() + *pole2->outPosition()) / 2;
    indicates if the middle of the two poles is near pole3
(2) VPort *bNear = *dist( *middle, ball->outPosition() ) < 10;
create software interactions that access virtual ports
(3) Event *evtl = new Event( middle, ball->inPosition() );
(4) CondEvent *cEvt = new CondEvent( bNear );
add interactions to activity
(5) cEvt->setTrueEvt( evtl );
(6) Activity *act = new Activity;
(7) act->addActEvent( cEvt );
\end{verbatim}

Figure 5.2: A usage of virtual ports that propagates backwards for values

\subsection*{5.2.2 Extended Example Revisited}

The code in Figure 5.4 modifies the code of the general example (Figure 3.1) to use three virtual ports: \texttt{hand}, \texttt{dist}, and \texttt{zero}. The three ports simplify and improve the code that determines whether the primary robot and the button touch. The three ports replace the software interactions \texttt{E} and (part of) \texttt{I}, and the software component \texttt{Distancer} of the general

\(^2\)Circular references arise if a virtual port initiates a search that returns to itself. Such instances are not permitted with the current implementation of interaction constraints. However, as noted in section 13.4.2, the resolution of circular references to interaction constraints is an area worthy of future investigation.
example, with three one-way constraints. The first constraint identifies a simple equation to determine the position of the robot's hand in world coordinates (line 11). Unlike the code of the original example, the two objects meet when the robot's hand, not its body, touches the button. The second constraint applies a function Dist for computing the distance between two ports (line 12). Dist provides the virtual port dist with an equation that dist evaluates when it receives a request for information. The final constraint identifies a logical relation between the port dist and the constant zero (line 13). When cevM requests a value, the final constraint initiates a chain of requests that propagates backwards to the first two virtual ports (line 28). From the response to these requests, the final constraint indicates clearly to cevM whether the distance between the robot's hand and the button is zero.

The diagram in Figure 5.5 illustrates the data flowing to and from the three virtual ports. When cevM executes, it requests a value from zero to determine if it is true or false. If zero is true, the software interaction initiates lightTog, an interaction that toggles the state of the light. zero satisfies the request by initiating its own request to dist and comparing the result to zero, a constant. dist responds by evaluating the equation provided by Dist and by requesting a value from hand. The final request to hand computes a value and terminates the backwards propagation for information. The entire process repeats each time that cevM requires a value from zero. Each request to zero accesses the most recent values for all ports involved. Thus, zero always reflects properly the meeting of the robot and button. The movements of either object do not create problems or require changes to zero's definition.

5.3 Virtual Events

A virtual event is a short-hand mechanism for expressing symbolic relationships among software interactions. Applying operators similar to that of virtual ports, a virtual event forms
CHAPTER 5. INTERACTION CONSTRAINTS

/* pass info from outport to import(s) */
(2) Event evA( rob->outPos(), pCtr->inBegin() );
(3) Event evB( rob->outHand(), hCtr->inBegin() );
(4) Event evC( pCtr->outVal(), rob->inPos() );
(5) Event evD( hCtr->outVal(), rob->inHand() );
(6) Event evE( rob->outHand(), dist->inPos2() );
(7) Event evF( rob->outHand(), grph->inXY(), filtXY );
(8) Event evP( rob->outPos(), sub->inPos1());
(9) Event evQ( sub->outPos(), hCtr->inEnd());
(10) FanEvent evfl( but->outPos(), Ctr->inEnd(), pCtr->inEnd() );
/* use virtual ports to determine if robot touches button */
(11) OPort *hand = *rob->outHand() + *rob->outPos();
(12) OPort *dist = Dist( hand, but->outPos() );
(13) OPort *zero = (*dist == 0);
... /* pass duration to imports */
(18) DurationEvent devX( hCtr->inDuration(), pCtr->inDuration() );
/* pass time step to imports */
(19) TimeEvent tevY( hCtr->inTime(), pCtr->inTime() );
... /* invoke import info if outport is true */
(28) CondEvent cevM( zero );
(29) cevM.setTrue( CallEvent( lght->inToggle() ) );
... /* organize software interactions */
(32) Activity act = new Activity;
(33) act->setInitEvt( evfl, devX, evA && evB );
(34) act->setActEvt( tevY, evC, evD, evE + evF, cevM, evP, evQ );

Figure 5.4: An implementation of the extended example that uses three virtual ports

a one-way constraint among software interactions that is satisfied each time the virtual event
receives a request for action from a script or another virtual event. Similar to the requests
for information that virtual ports receive, a request for action instructs a virtual event to
execute a behavior, as opposed to simply computing a (numerical) value. In addition, a
request for action divides into two kinds: executive or logical. An executive request initiates
a sequence of software interactions while a logical request examines an ordering of software
interactions and decides if the ordering is a recent occurrence.

5.3.1 Executive Request vs. Logical Request

An executive request constrains the execution of software interactions. The request applies
the operators of a virtual event to decide upon an ordering of software interactions. The
operators determine which interactions appear before others and which interactions execute
stochastically. The expression "vEv = *ev1 AND *ev2," for instance, indicates that the
interactions ev1 and ev2 execute before vEv. Had the expression applied "OR" instead of "AND," then either of the two interactions — but not both — would execute prior to vEv. In contrast, a logical request performs a query on the recent execution of particular software interactions. The query confirms if the interactions of the virtual event are currently active. For visual simulations, the interactions of a virtual event are active if they share the same execution times as those of the virtual event. The operators of the virtual event apply logical operations to the query performed on each interaction. For example, the expression "vEv = *ev1 OR *ev2" returns a positive response if either ev1 or ev2 occurred before vEv received a request for action.

The short code segment in Figure 5.6 exemplifies the production and application of virtual events in a script. It presents two virtual ports vEv1 and vEv2 (lines 2-3), as illustrated in Figure 5.7. vEv2 receives both types of requests for action. An executive request is initiated by the activity act. When act sequences through its set of interactions, established by line 7, it initiates vEv2 to activate either vEv1 or evt3. If the virtual event activates the former, the initial request propagates to vEv1, which responds by activating evt1 and evt2 in sequence. A logical request is performed by the conditional event cEvt (line 4). Upon execution, the interaction associates vEv2 with the activation of evt4. vEv2 returns a positive response if either vEv1 or evt3 precedes the activation of vEv2. Since vEv1 is a virtual event, it is considered to precede vEv2 if both of its constituents, evt1 and evt2, are already active.

5.3.2 Utility

A virtual event is useful for encapsulating a sequence of software interactions and for introducing variability to the sequencing of software interactions. A virtual event may apply
declares various events
(1) Event *evt1, *evt2, *evt3, *evt4;
defines virtual events
(2) VEvent *vEv1 = *evt1 && *evt2;
(3) VEvent *vEv2 = *vEv1 || *evt3;
employs virtual event as boolean signal
(4) CondEvent *cEvt = new CondEvent( vEv2 );
(5) cEvt->setTrueEvent( evt4 );
add interactions to activity
(6) Activity *act = new Activity;
(7) act->addActEvent( vEv2, cEvt );

Figure 5.6: An exemplary definition of two virtual ports

Figure 5.7: The virtual events of Figure 5.6

multiple relations, or reference the relations of other virtual events, to form and to encapsulate a lengthy sequence of constraints. To access the lengthy sequence, a script forwards a request to the virtual event that heads the sequence. To repeat the sequence, the script simply forwards another request to the same virtual event. The script avoids having to recreate any sequence that it needs to apply multiple times. A virtual event may also vary its use of relations so as to introduce variability to a lengthy sequence of constraints. As the virtual event performs, it applies statistical methods in evaluating its relations. As the relations vary, the virtual events vary the sequencing of communications.

5.3.3 Extended Example Revisited

The code in Figure 5.8 updates the code of Figure 5.4 to use three virtual events: vEv, vEv2 and ready (lines 25-27). The three events decide and influence the execution of cevM (line 28). Rather than directly accessing the virtual port zero for its logical condition, cevM accesses the virtual event ready, a logical expression that integrates the value of zero with the execution states of two events affecting the light’s position and orientation, mvLight and
rotLight.

```c
/* new events for extended example */
(11) Event mvLight( lposCtr->outVal(), lgght->inPos() );
(12) Event rotLight( loreCtr->outVal(), lgght->inOrient() );
(13) Event chgnCol( clrCtr->outClr(), lgght-MnColor() );
(14) CallEvent upClr( clrCtr->inChgClr() );
(15) CallEvent lightTog( lgght->inToggle() );
(16) ChainEvent chnEvts( upClr, chgnCol );
/* pass remaining duration to imports */
(17) RemDurEvent remDur( pCtr->inDuration() );
/* virtual events to constrain events */
(26) VirEvent *vEv = evA >> remDur;
(27) VirEvent *vEv2 = lightTog && (chnEvts || vEv);
/* invoke import info if output is true */
(28) OEvent *ready = zero && !(mvLight || rotLight);
(29) CondEvent cevM( ready );
(30) cevM.setTrue( vEv2 );
```

Figure 5.8: An implementation of the extended example that uses two virtual events

As illustrated in Figure 5.9a, the first two virtual events encapsulate two separate sequences of communications. `vEv` identifies a simple sequence involving two software interactions. In response to an executive request, `vEv` begins `evA` and finishes with `remDur`. `vEv2` identifies a parallel execution of two events, `lightTog` and either `chnEvts` or `vEv`. While `lightTog` executes, a statistical method invokes either of the two remaining interactions.

![Diagram](A) Executive

![Diagram](B) Logical

Figure 5.9: The virtual events of the extended example

The virtual event `ready`, as illustrated in Figure 5.9b, identifies a logical expression with the virtual port `zero` (see Figure 5.4) and the software interactions `mvLight` and `rotLight`. `ready` is true when `zero` evaluates to true and neither of the two software in-
teractions produces communications. In other words, the sequence of events that surround the toggling of the light occurs only when the robot touches the button and the light is stationary.
Chapter 6

Multi-Modeling Methods

The separation between computation and coordination, offered by the use of software interactions, presents a variety of opportunities for applying computational algorithms to the coordination of communications. The methods of world view modeling [169], for instance, augment IBP with the algorithms of computer simulation. Rather than updating the state of a program directly, the methods of world view modeling, such as finite state automata and Petri nets [116], work readily in coordinating the workings of software interactions. As the algorithms vary the execution of software interactions, the components of a program send and receive information to update the computational state. The scripts and meta-scripts of IBP freely apply these algorithms in producing programs effectively and in applying software composition to the building of code.

With the introduction of computational algorithms to IBP, the software interactions of IBP undertake various forms. Those forms separate into several categories, each of which classifies a different behavior. This chapter begins with a discussion of the working forms of software interactions and follows with a presentation of the various uses of computational algorithms in updating the design of scripts and in applying software composition to the building of IBS programs.

6.1 Forms of Software Interactions

In applying the computational methods of computer simulation to IBP, the software interactions of IBS develop into four distinct categories: communicational, temporal, organizational, and informational. Communicational interactions communicate information, temporal interactions synchronize components, organizational interactions collate interactions, and informational interactions support debugging. Augmenting any of these interactions with
time-varying information produces a *time-varying interaction*. The information specifies when, how often, and under what conditions a software interaction executes.

### 6.1.1 Categories

The information in Table 6.1 identifies the four categories of software interactions and presents examples of each. Each software interaction is presented with a number that indicates how many ports it accesses during its execution or how many other interactions it manages.

<table>
<thead>
<tr>
<th>Kind</th>
<th>SubKind</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communicational</td>
<td>Simple</td>
<td>Event(2); FanEvent(&gt;2), StateEvent(1); CallEvent(1); DataEvent(1); BoolEvent(1); CondEvent(1); BiCondEvent(2); WhileEvent(1); ChangeEvent(1)</td>
</tr>
<tr>
<td>Logical</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temporal</td>
<td>Time</td>
<td>TimeEvent(1); StepEvent(1); ReStepEvent(1); DurationEvent(1); RemainDurEvent(1)</td>
</tr>
<tr>
<td>Functional</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Organizational</td>
<td>Group</td>
<td>ToggleEvent(<em>); ChainEvent(</em>); SwitchEvent(<em>); RandToggleEvent(</em>); RandChainEvent(<em>); RandSwitchEvent(</em>);</td>
</tr>
<tr>
<td>Random</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Informational</td>
<td>Trace</td>
<td>TextEvent(1); DebugEvent(1)</td>
</tr>
</tbody>
</table>

Table 6.1: Various kinds of events

Communicational software interactions pass information among the ports of software components. They simply ensure that information passes properly from a single outport to one or more inports. Communicational software interactions divide into two sub-groups: *simple* and *logical*. Simple communicational interactions perform basic actions, such as passing information among ports (*EVENT*) or invoking ports to induce change (*CALL EVENT*). They accept the use of filters to modify the passing of information among components. Logical software interactions, such as *BOOL EVENT* and *WHILE EVENT*, evaluate logical values (from logical outports) to produce communications. The logical values determine what the interactions produce, and for some, how often the interactions operate.

Temporal software interactions synchronize software components with the advancement of time. They determine when software components update state and how often the updates occur. Temporal software interactions divide into two sub-groups: *time* and *functional*. Time interactions describe and control the advancement of time. Some interactions, such as *TIME EVENT* and *STEP EVENT*, identify the current time and its advancement. Others, such as *DURATION EVENT* and *REMAIN DURATION EVENT*, identify the duration of their
executing script and the time remaining before the script ends. Functional interactions modify the execution of their governing action unit. They modify their script to wait for time to pass (\texttt{HOLDEVENT}), to await a logical value (\texttt{WAITEVENT}), or to await a signal to resume execution (\texttt{SUSPENDEVENT}).

Organizational software interactions arrange other software interactions into primitive structures. Unlike the mechanisms that script the execution of software interactions, the primitive structures neither sequence software interactions nor establish attribute values. The structures generally manage the software interactions as a single structure with one or more operational modes. Organizational software interactions divide into two sub-groups: \textit{group} and \textit{random}. Group interactions, such as \texttt{CHAINEVENT} and \texttt{SWITCHEVENT}, collate collections of software interactions into a single unit. When executed, the single unit selectively decides which of its software interactions to trigger. Random interactions operate similarly to group interactions. They perform the same action, but with some randomness. For example, \texttt{RANDSWITCHEVENT} decides randomly which event to trigger from its collection of interactions.

Informational software interactions communicate the values of (out)ports. Usually for debugging purposes, they write information to a file or an output device. Informational software interactions consist of one sub-group: \textit{trace}. Trace interactions identify the run-time properties of an IBS program. Primarily for debugging purposes, they produce information to a file or an output device. For example, \texttt{DEBUGEVENT} presents the states of several outports. As the program executes, the software interaction presents a periodic history of state information.

### 6.1.2 Time-Varying

Especially useful for visual simulation, a time-varying interaction binds information with another interaction so as to vary the use of interactions. This information delays the execution of a software interaction until specific times pass. Unlike a software interaction with temporal attributes, a time-varying interaction separates easily into two parts, an interaction and a specification of time. In many instances, the development and maintenance of each part occurs separately. In an interaction with temporal attributes, the temporal information is set and normally is only one specific kind or value.

Since a time-varying interaction develops from parts, it supports the reuse of temporal information. Temporal information may be separated from one interaction and bound to another. For animation and other dynamic systems, reuse of temporal information is as important as the reuse of interactions and components. The nature of a scene is characterized
equally by the temporal information of its interaction and by the communications of its software components. Binding an interaction with the temporal information of another produces a scene with temporal characteristics identical to those of the other. The actions and results of the two scenes may differ, but their temporal characteristics are the same. Similarly, updating the temporal information of an existing interaction produces a scene similar in action to the original. The two scenes share the same actions but differ in their temporal characteristics.

6.1.3 Extended Example Revisited

The diagram in Figure 6.1 details the diagram of Figure 3.2 with twelve software interactions: nine communicational and three temporal. The communicational interactions pass information among the robots, light, and controllers. Consisting of four EVENTS, two FANEVENTS, two CALLEVENTS, and one CONDEVENT, the communicational interactions configure the software network to associate the light's position, orientation, and color with the controllers. Many of the interactions, such as evFI but and upClr, provide the controllers with initial control values. With those values, the controllers compute an appropriate location, orientation, and color, which mvLight, rotLight, and chngClr communicate to the light. The temporal interactions synchronize the controllers to the advancement of time. Consisting of one DURATIONEVENT, one REMAINDUREVENT, and one TIMEEVENT, the temporal interactions permit the controller to know how fast time advances and how much time the controller should expect to complete its actions. For example, the software interactions X and Y decide how long and how often POSCONTROL and ORIENTCONTROL produce positional and orientational values. The periodic execution of the communicational interactions determines when and how often the values reach the light.

6.2 Scripts

With the introduction of computational algorithms to IBP, the scripts of IBS divide into four distinct kinds: natural, declarative, functional, and group. Each kind of script outlines an alternative plan to synchronize software interactions to the passage of time.

---

1 The software components and software interactions for updating the positions and orientations of the robots appear in Figure 4.3.

2 The FANEVENTS of the Figure 6.1 also interconnect the components of the original example (see Figure 3.2).
6.2.1 Kinds of Scripts

The information in Table 6.2 identifies four kinds of scripts and presents two variants of each. The two variants, single and multiple, indicate the number of threads of execution that each script controls. A single-threaded script executes one set of interactions. Any changes to the ordering or execution of the interactions apply equally to the entire set. A multiple-threaded script executes multiple sets of interactions. Each set is managed as an individual group that does not affect the execution of others.

<table>
<thead>
<tr>
<th>Kind</th>
<th>single</th>
<th>multiple</th>
</tr>
</thead>
<tbody>
<tr>
<td>Natural</td>
<td>Activity; RandAct</td>
<td>MultiActivity</td>
</tr>
<tr>
<td>Declarative</td>
<td>DetFSAutAct; NonFSAutAct; PetriAct</td>
<td>MultiProcess</td>
</tr>
<tr>
<td>Functional</td>
<td>Process</td>
<td>MultiGroupAct</td>
</tr>
<tr>
<td>Group</td>
<td>GroupAct; ToggleAct; SwitchAct; RandGrpAct</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: Various kinds of scripts

Natural scripts order software interactions sequentially. They repetitively parse and execute an entire sequence of software interactions. The code of Figure 6.2 presents the use of MULTIACITY, a natural script with several threads of control. Individual threads are identified by the constant THREAD_#, where # corresponds to a particular thread. When the script begins, it simultaneously executes two threads of initial events (lines 2-3). Afterwards, it continuously sequences through the two threads of acting events (lines 4-5).
CHAPTER 6. MULTI-MODELING METHODS

Declarative scripts apply declarative methods, such as event graphs [135], to partition the ordering and execution of software interactions into several states. Each state maintains a distinct list and ordering of the software interactions. For each state, there are rules that indicate how it arises and when it changes. Collectively, the rules establish a total ordering of state space. As the rules advance the script from one state to the next, they initiate a new ordering and execution of software interactions.

Functional scripts use functional methods, such as those performed by SIM++ [90], to regulate the execution of software interactions. The scripts work in conjunction with functional software interactions (section 6.1.1) to regulate the flow of information between software components. Information flows according to the satisfaction of operational constraints, such as the availability of resources. Functional scripts best model systems in which a directionality of flow exists between system components, such as queues and control engineering.

Group scripts collate collections of scripts into a single unit. When executed, the single unit selectively decides which of its scripts to apply. For instance, the code appearing in Figure 6.3 presents two group scripts, GROUPACT and RANDGrpAct. Each group script assembles the same set of scripts but applies the scripts differently. GROUPACT executes every script while a RANDGrpAct executes only one, which is chosen at random. The behavior of the former script is deterministic while the latter is not.
6.2.2 Declarative

Declarative methods, which normally govern the ordering of state changes, apply equally well to the scheduling of software interactions. Rather than issuing commands to update state, the methods are easy to apply to decide how and why software interactions operate. The scheduling of software interactions occurs as the declarative methods evaluate transition functions, conditional statements that alter a method’s behavior. The transition functions and the modeling methodology they represent may be encapsulated within abstractions that accept software interactions and ports as parameter values. When the abstractions are executed, they interpret their transition functions and schedule an appropriate response.

```plaintext
produces sequences of values
(1) EvolutePt *ev = new EvolutePt;
    interactions to parameterize interpolator
(2) Event *eSet = new Event( ball->outPosition(), ev->inBeginVal() );
(3) Event *ef1 = new Event( pole1->outPosition(), ev->inFinishVal() );
(4) Event *ef2 = new Event( pole2->outPosition(), ev->inFinishVal() );
(5) Event *ef3 = new Event( pole3->outPosition(), ev->inFinishVal() );
    interactions to compute new position
(6) DurationEvent *durEvt = new DurationEvent( ev->inDurationVal() );
(7) TimeEvent *tEvt = new TimeEvent( ev->inCalcValue() );
(8) Event *bSet = new Event( ev->outCurVal(), ball->inPosition() );
    add interactions to a finite state automaton script
(9) FiniteStateAct *fsa = new FiniteStateAct( 1 );
(10) fsa->setState( 1, eSet, ef2 );
(11) fsa->setState( 2, eSet, ef3 );
(12) fsa->setState( 3, eSet, ef1 );
(13) fsa->setTransition( 1, 12, 2 );
(14) fsa->setTransition( 2, 12, 3 );
(15) fsa->setTransition( 3, 12, 1 );
    add interactions to an activity
(16) Activity *act = new Activity;
(17) act->addInitEvent( durEvt );
(18) act->addActEvent( fsa, tEvt, bSet );
```

Figure 6.4: A script that moves a ball between three poles

The sample code in Figure 6.4 exemplifies the application of declarative methods to the organization of software interactions. The code consists of seven events and two scripts, of which one is `fsa`, a script specifying a finite state automaton (FSA) — a common abstraction of declarative modeling. Placed within the script `act`, the FSA (lines 10-16) controls interactions for moving a ball towards one of three poles. When the script is active, `act` continuously sequences through `fsa`, `tEvt`, and `bSet` (line 18). Every twelve units of
time, \( f_{sa} \) adjusts the motion of the ball to advance toward a different pole. The illustration in Figure 6.5 presents the state description of \( f_{sa} \). As \( f_{sa} \) advances from one state to the next, it activates a different set of software interactions (lines 11-13).

![Figure 6.5: The finite state automaton of Figure 6.4](image)

The illustration in Figure 6.6 visualizes the software interactions of Figure 6.4 and identifies their association with the states of the FSA. All three states of the FSA share the interactions \( t_{Evt}, e_{Set}, \) and \( b_{Set} \). Each of the three interactions communicate information to or from \( ev \), a controller for interpolating positional data. \( t_{Evt} \) communicates temporal advancement, \( e_{Set} \) communicates an initial position from which the controller begins interpolation, and \( b_{Set} \) communicates a positional value from \( ev \) to the ball. Each of the three states of the FSA are distinguished by the interaction that sets the terminal value of \( ev \). The terminal value is set to one of three pole positions by the interactions \( ef_{1}, ef_{2}, \) and \( ef_{3} \).

![Figure 6.6: A visualization of the script in Figure 6.4. The numbers adjacent to the software interactions correspond to the states of the FiniteStateAct of Figure 6.5](image)
6.2.3 Functional

Functional methods, which normally regulate dataflow, also prove useful in controlling the evaluation of software interactions. Rather than regulating communications, the methods decide when and why software interactions undergo evaluation. They determine how the interactions of a script, as determined by declarative methods, are evaluated in relation to each other and during the progression of information among components. For most functional methods, the basic abstraction is the process, a sequence of instructions that work collectively to perform a task. The instructions either define a succession of state changes or modify the manner in which its governing process interprets subsequent instructions. Those instructions, which may be encapsulated in interactions that manage ports, may be adapted to update the manner in which a script processes its interactions.

At any moment in time, a process is active, idle, holding, or waiting. As described in Table 6.3, an active process actively executes its instructions, an idle process awaits (re)activation, a holding process awaits a state change, and a waiting process awaits the passage of time. The performance of a process changes as interesting events occur, such as the presence or absence of information, or the notification of state changes. A process switches among the four states according to the dictates of its most recent instruction. For example, a waiting instruction directs the process to halt temporarily its operations for a short period of time. That same instruction may be adapted to halt the execution of a script as it processes a set of software interactions. Instructions of that kind, identified as functional events (see section 6.1.1), provide careful controls in coordinating a wide variety of communications.

<table>
<thead>
<tr>
<th>Process State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>active</td>
<td>actively executing instructions</td>
</tr>
<tr>
<td>holding</td>
<td>awaits an interesting event or state change</td>
</tr>
<tr>
<td>waiting</td>
<td>awaits the passage of time</td>
</tr>
<tr>
<td>idle</td>
<td>inactive; awaits re-activation</td>
</tr>
</tbody>
</table>

Table 6.3: Description of process states

The sample code in Figure 6.7 exemplifies the application of functional methods to the organization of software interactions. The code controls the evaluation of software interactions by referencing two operational constraints, the presence of information (lines 2-3) and the passage of time (lines 6-8). The code operates by evaluating five events for four components: SINK, SOURCE, QUEUE, and WAITTIME. Information passes from SOURCE to SINK via QUEUE when WAITTIME, a component that produces exponential values, indicates
that it should do so. The event srcEvt indicates that the script should wait for SOURCE to produce information before the script handles additional events. When SOURCE has output available, a software interaction occurs and the evaluation of the script resumes. The event snkEvt acts similarly. It halts the evaluation of the script until SINK is available to accept information. The events cEvt and tEvt decide how much time passes between the first two events. The temporal value is obtained from the exponential.

```
create a queue to hold data
(1) FifoQueue *queue = new FifoQueue;

moving data from source to queue
(2) WaitEvent *srcEvt = new WaitEvent( source->outHasOutputAvail() );
(3) srcEvt->setTrueEvt( source->outExtract(), queue->inEnqueue() );

move data from queue to sink
(4) WaitEvent *snkEvt = new WaitEvent( sink->outIsAvail() );
(5) snkEvt->setTrueEvt( queue->outDequeue(), sink->inInput() );

compute waiting time
(6) ExponentialDbl *waitTime = new ExponentialDbl(10);
(7) CallEvent *cEvt = new CallEvent( waitTime->inMakeVal() );
(8) HoldEvent *tEvt = new HoldEvent( waitTime->outVal() );

organize events into a process
(9) Activity *prc = new Activity( "queuing" );
(10) prc->addActEvent( srcEvt, cEvt, tEvt, snkEvt, tEvt );
(11) return prc;
```

Figure 6.7: A script that moves information between SINK and SOURCE via QUEUE.

The organizational pattern of the events decided by the activity (line 10) partitions the evaluation of the script into four phases, as illustrated in Figure 6.8. In phase one, the script awaits information to be passed from SOURCE to QUEUE. In phase two, the script awaits the passage of time. In phase three, the script awaits information to be passed from QUEUE to SINK. In phase four, the script waits for the same amount of time that it waited for in phase two. The script continually repeats all four phases until it is terminated by other abstractions (see [86, 87] for details).

![Figure 6.8: The four phases of Figure 6.7](image-url)
CHAPTER 6. MULTI-MODELING METHODS

6.2.4 Extended Example Revisited

The code in Figure 6.9 completes the coding of the extended example with two scripts, *fsaAct* (lines 21-24) and *act* (lines 32-36). *fsaAct* describes the communications of the original robot with a "2-state" finite state automaton (FSA) while *act* handles the actions of the robot and light with a "3-thread" multi-threaded process.

---

```c
/* determines the meeting of the two robots */
(20) OPort *meet = (*rob->outPos() == *rob->outPos());
/* finite state automata activity */
(21) FsaActivity fsaAct;
(22) fsaAct.setState(0, evFI.but, evC && evD);
(23) fsaAct.setState(1, evFI.rob, evC && evD);
(24) fsaAct.setTransition(1, 0, meet || 60);
(25) CallEvent trnsEvt( fsaAct.inTransition(0, 1));
/* virtual events to constrain events */
(26) VirEvent *vEv = trnsEvt >> evA >> remDur;
(27) VirEvent *vEv2 = light Tog (chnEvts || vEv);
/* invoke import info if outport is true */
(28) OEvent *ready = zero !(mvLight || rotLight);
(29) CondEvent cevM( ready);
(30) cevM.setTrue( vEv2);
/* time delay events */
(31) WaitEvent wEvt10( 10 ), wEvt20( 20 );
/* organize software interactions */
(32) MultProcess *act = new MultProcess;
(33) act->setInitEvt( THREAd.0, evFl, devX, evA && evB);
(34) act->setActEvt( THREAd.0, tevY, fsaAct, evD, evF, cevM);
(35) act->setActEvt( THREAd.1, wEvt10, mvLight, wEvt10);
(36) act->setActEvt( THREAd.2, wEvt20, rotLight);
```

---

Figure 6.9: An implementation of the extended example that applies two scripts

Each of the states of *fsaAct* describes a different set of interactions for controlling the robot's movements. In state zero (line 22), the interaction *evFI.but* updates POSITION-CONTROL to advance the robot towards the button. In state one (line 23), the interaction *evFI.rob* updates the same controller to move the same robot towards *robotB*. Beginning in state zero, the script switches respectively between the two states according to two transitions (lines 24-25). State one occurs when either the virtual port *meet* achieves a positive value.

---

The working version of the code segments permits some latitude in deciding when the two robots meet. For brevity's sake, the statements are approximated with the definition of line 20.
or sixty units of time pass. State zero re-occurs when the CALLEVENT trnsEvt executes, as performed by the newly updated definition of the virtual event vev (line 25).

Each of the threads of act regulates a separate task. The first thread, THREAD_0, nearly replicates the organizational design of the script of Figure 5.4. Instead of directly accessing evC, which passes information from POSITION-CONTROL to the robot, the script invokes fsaAct. When fsaAct executes, it determines when and how to execute evC. The second thread, THREAD_1, intermixes WaitEvent with an interaction to move the light. The repetitive execution of the thread leads to periodic movements to the light's position. When the thread begins, it waits for 10 units of time. Then, it executes mvLight and re-waits for another 10 units of time. The third thread, THREAD_2, intermixes WaitEvent with an interaction to update the orientation of the light. Similarly to THREAD_1, THREAD_2 waits 20 units of time between successive calls. The ordering of the two threads differs by their offset. THREAD_1 initially executes 10 units of time earlier.

The organization and structures of the two scripts provide several benefits to program development. First, the application of multiple states and threads establishes relationships among sequences of software interactions. The sequences relate transitionally, when formed with states, or concurrently, when formed with threads. Second, the distinct representation of states and threads facilitates the development and reuse of sets of software interactions. Each state or thread may be individually inserted or removed from a script with little difficulty. Third, the application of functional methods and their functional events, such as WaitEvents, is practical for establishing temporal delays among software interactions. The temporal delays that separate the execution of software interactions greatly affect the behavior and output of a system.

The complete code for the extended example appears in Figure 6.18. The code forms a script for advancing a robot towards a light and having the robot toggle the light's state. The entire script consists of seventeen software interactions (events), three virtual ports, three virtual events, and two activities.

6.3 Software Composition

With computational algorithms providing a foundation upon which to build scripts, IBP supports software composition, the building of software with the systematic integration of building blocks, abstractions that encapsulate functions and commands. The organization and binding of building blocks determines how a software system behaves and what information it contains. With software interactions as building blocks, software composition proves
useful in supporting the development of multi-model scenes, visual simulations operating under the influence of several modeling methods such as those discussed in the previous section.

Two popular forms of software composition, as noted by Fishwick [46], are aggregation and assembly. Aggregation builds a large block from collections of smaller blocks while assembly produces a new block from the combination of existing blocks. Aggregation produces a block that is identified by its parts while assembly produces a block with its own identity. Each of the techniques applies well to the development of high-level abstractions and the reuse of IBS. Aggregation forms a script from multiple models while assembly transforms a script into a block resembling a software component.

6.3.1 Aggregation

In a script, the aggregation of software interactions develops as a linear ordering or a hierarchical grouping. A linear ordering organizes a sequential arrangement of software interactions while a hierarchical grouping organizes a ranked order. Within a linear ordering, software interactions develop as a single abstraction by intertwining or integrating sequentially. When the interactions intertwine, they execute several modeling methods simultaneously. When the interactions integrate sequentially, they execute individual modeling methods in tandem. Within a hierarchical grouping, software interactions embed hierarchically. Interactions of one set operate under the influence of another set. Each level of a hierarchical grouping applies a different level of abstraction in aggregating software interactions. Each level refines the interactions of the previous level.

When a hierarchical aggregation integrates multiple modeling methods, the resulting aggregation forms a heterogeneous multi-model script. Each part of the aggregation employs the modeling method that best matches its requirements. When the parts apply similar methods, the parts assemble a homogeneous multi-model script. The entire aggregation employs one method to organize several sets of interactions. A homogeneous multi-model script is similar but not equivalent to a linear aggregation with multiple parts. A homogeneous multi-model script models from a single abstraction level while a linear aggregation models from multiple abstraction levels, all of which happen to be the same.

The illustration in Figure 6.10 exemplifies the use of the two forms of aggregation in developing one multi-model system. The joining of \( P \) with \( Q \) illustrates a linear ordering: \( Q \) follows the operations of \( P \). The placement of \( Z \) within \( P \) represents a hierarchical grouping. \( Z \) refines the definition of \( P \) homogeneously: both \( Z \) and \( P \) employ the same organizational method.
6.3.2 Assembly

The grouping of software interactions into a script supports the building of an assembly, an abstraction that behaves in a manner similar to that of a software component. An assembly has identity and supports ports as member functions. The ports establish an interface for the assembly to interact with its environment. Incoming information parameterizes the behavior of the script while outgoing information presents the results of parameterization. With ports, the assembly operates freely with software interactions in communicating information. It readily consumes and produces information as it sends and receives. Behaving similarly to a software component, an assembly actively participates in the formation of other scripts. An assembly may be progressively assembled to produce scripts of greater design and function. Despite the similarities, an assembly's internal design differs from that of a traditional software component. An assembly is an abstract entity for coordinating communications, not for performing computational algorithms. It is a scripted configuration that acts out the role of a software component.\(^4\)

There are several benefits of producing an assembly from a script. First, the assembly facilitates software development by composition. The coordination of communications among software components does not stop with the definition of a script. As a software component, the script freely interacts with other software interactions to build other scripts of greater size and complexity. Second, the assembly encourages modular development of IBS programs. As a software component, a script may be repeatedly invoked to form other scripts of varying designs. Changes to the original script are localized and hidden from its users. Third, the assembly promotes an alternative means of reusing a script. Rather than

\(^4\)The question of whether the software guidelines of section 8.3 apply equally to both software components and assemblies is under investigation.
extracting its parts or embedding the code directly into another script, a script may be set aside as a software component ready for reuse. Users search for and integrate the scripts or software components that meet their needs.

```c
ActivityComp* helicMotion()
{
    produces sequences of values
    (1) EvHelixPt *ev = new EvHelixPt();
    identifies two proxy ports
    (2) InProxy *iProxy = new InProxy( "in" );
    (3) OutProxy *oProxy = new OutProxy( "out" );
    virtual ports sum incoming proxy and spinPt's value
    (4) VPort *newPos = *ev->outCurVal() + *iProxy->outVal( "in" );
    interactions to compute new position
    (5) DurationEvent *durEvt = new DurationEvent( ev->inDurationVal() );
    (6) TimeEvent *evt0 = new TimeEvent( ev->inCalcValue() );
    (7) Event *eSet = new Event( newPos, oProxy->inVal( "out" ) );
    add interactions to a coordination activity
    (8) ActivityComp *act = new Activity( "spin" );
    (9) act->addInitEvent( durEvt );
    (10) act->addActEvent( evt0, eSet );
    (11) return act;
}
```

Figure 6.11: An assembly for producing a helical motion

The code in Figure 6.11 exemplifies the production of an assembly that computes helical motion. The assembly augments a script with two proxy ports, iProxy and oProxy. The proxy ports establish an external interface for the assembly. As illustrated in Figure 6.12, iProxy accepts an incoming value while oProxy emits an outgoing value. The software interactions of the assembly execute each time iProxy receives a new value. The last software interaction, eEvt, provides oProxy with an outgoing value. The virtual port (see section 5.2), which provides eEvt with information, computes the results of parameterization by adding the incoming parameter value with the outgoing value of ev, a software component for computing helical values.

**Applying an Assembly**

The code in Figure 6.13 exemplifies the application of the assembly by another script that moves a ball toward a pole. The assembly updates the linear movements of the ball to produce a helical motion. The script interfaces with the assembly by accessing the proxy ports of the assembly by name (lines 7-8). The "in" proxy port obtains a parameter value
CHAPTER 6. MULTI-MODELING METHODS

from the motion interpolator ev via the event ev0 while the “out” proxy port returns a helical value that the event eSet uses to set the ball’s position.

Activity* moveHelically( Model *ball, Model *pole )
{
    obtains assembly
    (1) ActivityComp *hlMv = helicMotion();
    produces a sequence of values
    (2) EvolutePt *ev = new EvolutePt();
        interactions that move a ball towards a pole
    (3) Event *efB = new Event(ball->outPosition(), ev->inBeginVal());
    (4) Event *efF = new Event(pole2->outPosition(), ev->inFinishVal());
    (5) DurationEvent *durEvt = new DurationEvent(ev->inDurationVal());
    (6) TimeEvent *evtT = new TimeEvent(ev->inCalcValue());
        interactions that interface with the assembly
    (7) Event *evt0 = new Event(ev->outCurVal(), hlMv->inProxy("in"));
    (8) Event *eSet = new Event(hlMv->outProxy("out"), ball->inPosition());
        add interactions to an activity
    (9) Activity *act = new Activity("moveFSA3");
    (10) act->addInitEvent(durEvt, efB, efF);
    (11) act->addActEvent(evtT, evt0, eSet);
    (12) return act;
}

Hierarchical Assemblies

An assembly may reference other assemblies to produce a hierarchical assembly. As illustrated in Figure 6.14, the hierarchical assembly consists of multiple assemblies, all of which are bound by additional software interactions.\(^5\) As information flows into the hierarchical

\(^5\)The diagram of Figure 6.14 is unrelated to the diagram of Figure 6.12
assembly via the proxy ports, the software interactions incite communications and establish the values of outgoing information, which the proxy ports to others provide upon demand.

6.3.3 Extended Example Revisited

As illustrated in Figure 6.16, the code in Figure 6.15 partially reuses the code of the extended example to produce an assembly that accepts two references, one for a robot and another for a light. The assembly determines if the robot successfully moves to a specific location when the light is stationary. Specified by a proxy port (line 1), the value of the location is determined when the assembly is in use.

In reusing the original code, the assembly duplicates the virtual ports determining contact (lines 3-5), and converts as additional virtual ports (lines 6-7) the virtual events determining light movements. The conversion (reasonably) replaces the intent of the virtual events without referencing other events, as specified by the original coding. The conversion of the virtual events is performed with the port-producing function prevState. The function accepts a port as an argument and returns a new port returning the original port's most recently requested value. The virtual ports moving and rotating (lines 6-7) use prevState to compare the current and previous values of the light's ports that indicate position and orientation. A change in either value indicates that the light is moving or rotating. To reuse the original code exactly without changes would require that the assembly also accept references to software interactions, such as evA and remDur, that were accessed by the original virtual events. Rather than reacting to the execution of a software interaction, the new code reacts to the changes of state. Since software components are time-invariant, it is safe to infer an equivalence between a state change and the execution of a software interaction. State changes occur only after a software interaction invokes an inport to consume information. The concluding code for the assembly forms an interaction and a script (line 9-11). Together,
Activity* robotTouchWhileLightFixed( Robot *rob, Light *light )
{
    /* create assembly by identifying two proxy ports */
    (1) InProxy  *iProxy = new InProxy( "in" );
    (2) OutProxy *oProxy = new OutProxy( "out" );
    /* virtual ports to check if robot touches position identified by inProxy */
    (3) OPort   *hand = *rob->outHand() + *rob->outPos();
    (4) OPort   *dist = Dist( hand, iProxy->outVal( "in" ) );
    (5) OPort   *zero = (*dist == 0);
    /* virtual ports to determine if light neither moving nor rotating */
    (6) OPort   *moving = (prevState( light->outPos() ) == light->outPos());
    (7) OPort   *rotating = (prevState( light->outOrient() ) == light->outOrient());
    (8) OPort   *ready = zero && !(moving || rotating);
    /* pass boolean to outProxy */
    (9) Event   *eSet = new Event( ready, oProxy->inVal( "out" ) );
    /* add event to activity */
    (10) Activity *act = new Activity;
    (11) act->addActEvt( eSet );
    (12) return act;
}

Figure 6.15: An assembly formed from the script of the extended example

yield extract information from the virtual ports and prepare the information for the outgoing proxy port (line 2).

Figure 6.16: The assembly of Figure 6.15

The code segment in Figure 6.17 applies the assembly of Figure 6.15. It produces two instances of the assembly by invoking robotTouchWhileLightFixed twice. Each call acts upon the same light but with a different robot, rob or robZ (lines 27-28). Each instance receives information from an incoming proxy port, as indicated by touchA for
rob and by touchZ for robZ (lines 29-30). The two events provide a positional location either to actCompA or actCompZ, the two assemblies. The virtual port anytouch establishes a logical constraint between the outgoing values of each assembly (line 29). When the program executes, the ConDEvent cevM accesses the virtual port to decide whether it should execute the virtual event vEv2. The last two lines of the code segment (lines 33-34) create the script act that aggregates the methods of the two assemblies with a sequence of software interactions. The resulting script forms a hierarchical aggregation of software interactions. The immediate software interactions of the script (line 32-33) are high-level while those of the assemblies are low-level.

```c
/* using assembly */
(27) ActivityComp *actCompA = robotTouchWhileLightFixed( rob, lght );
(28) ActivityComp *actCompZ = robotTouchWhileLightFixed( robZ, lght );
/* does either robot touch button */
(29) Event *touchA = new Event( but->outPos(), actCompA->inProxy("in");
(30) Event *touchZ = new Event( but->outPos(), actCompZ->inProxy("in");
/* does any robot touch as light is stationary */
(31) OPort *anytouch = (actCompA->outProxy("out") ||
                        actCompZ->outProxy("out");
(32) CondEvent cevM( anytouch );
(33) cevM.setTrue( vEv2 );
/* form script */
(34) MultiActivity *act = new MultiActivity;
(35) act->setActEvt( THREAD_0, tevY, fsaEvt, evD, evF, touchA, touchZ, cevM );
```

Figure 6.17: A usage of the assembly for the extended example
Activity* robotTouchLightButton()
{
    /* pass info from output to inport(s) */
    (1) Event evA( rob->outPos(), pCtr->inBegin() );
    (2) Event evB( rob->outHand(), hCtr->inBegin() );
    (3) Event evC( pCtr->outVal(), rob->inPos() );
    (4) Event evD( hCtr->outVal(), rob->inHand() );
    (5) Event evF( rob->outHand(), grph->inXY(), filtXY() );
    (6) FanEvent evfLbut( but->outPos(), hCtr->inEnd(), pCtr->inEnd() );
    /* virtual ports to determine if robot touches button */
    (7) OPort *hand = rob->outHand() + rob->outPos();
    (8) OPort *dist = Dist( hand, but->outPos() );
    (9) OPort *zero = (*dist == 0);
    /* new events for extended example */
    (10) FanEvent evflrob( rob->outPos(), pCtr->inEnd(), pCtr->inEnd() );
    (11) Event mvLight( iposCtr->outVal(), lght->inPos() );
    (12) Event rotLight( loreCtr->outVal(), lght->inOrient() );
    (13) Event chgnCol( clrCtr->outClr(), lght->inColor() );
    (14) CallEvent upClr( clrCtr->inChgClr() );
    (15) CallEvent lightTog( lght->inToggleButton() );
    (16) ChainEvent chnEvts( upClr, chgnCol );
    /* pass duration to inports */
    (17) RemDurEvent remDur( pCtr->inDuration() );
    (18) DurationEvent devX( hCtr->inDuration(), pCtr->inDuration() );
    /* pass time step to inports */
    (19) TimeEvent tevY( hCtr->inTime(), pCtr->inTime() );
    /* determines the meeting of the two robots */
    (20) OPort *meet = ( *rob->outPos() == *rob->outPos() );
    /* finite state automata activity */
    (21) FsaActivity fsaAct;
    (22) fsaAct.setState( 0, evfLbut, evC & evD );
    (23) fsaAct.setState( 1, evfLrob, evC & evD );
    (24) fsaAct.setTransition( 1, 0, meet || 60 );
    (25) CallEvent trnsEvt( fsaAct.inTransition(0,1) );
    /* virtual events to constrain events */
    (26) VirEvent *vEv = trnsEvt + evA + remDur;
    (27) VirEvent *vEv2 = lightTog & (chnEvts || vEv);
    /* invoke import info if outport is true */
    (28) OEvent *ready = zero & !(mvLight || rotLight);
    (29) CondEvent cevM( ready );
    (30) cevM.setTrue( vEv2 );
    /* time delay events */
    (31) WaitEvent wevt10( 10 ); wevt20( 20 );
    /* organize software interactions */
    (32) MultiProcess *act = new MultiProcess;
    (33) act->setInitEvt( THREAD_0, evfl, devX, evA & evB );
    (34) act->setActEvt( THREAD_0, tevY, fsaAct, evD, evF, cevM );
    (35) act->setActEvt( THREAD_1, wevt10, mvLight, wevt10 );
    (36) act->setActEvt( THREAD_2, wevt20, rotLight );
    (37) return act;
}

Figure 6.18: Complete Listing of Code for Extended Example
Chapter 7

General Specification

Scene modeling is the building of virtual scenes with a specification that describes the shape and material characteristics of geometric forms. Scene modeling is widely applied to develop the many aspects of interactive applications that are either unchanging or reactive to user inputs. Software interactions and the organizational methods of this thesis expand existing specifications for scene modeling, in particular VRML [18] and JAVA3D [33], with a systematic approach to animating state. This approach, embodied as a general specification (for scene animation), develops visual simulations from an organization of parts, and encourages the development of a general file format for visual simulation that is logical, extensible, and device-independent.

In developing visual simulations by parts, developers apply a consistent approach to assemble and disassemble the production of animated scenes. Developers test and update their scenes by tweaking their parts, modifying their assembly, or introducing enhancements. Through using a general file format for visual simulation, developers produce data files that are consistent and easy to share. Visual simulations quickly exchange animations and apply them in new and alternative contexts.

Building upon IBP and its enhancements, the general specification consists of three hierarchical graphs: model, action, and time. The organization of the three graphs and their respective parts determines what appears and what occurs in visual simulations over time. The model graph extends the “scene graph,” the common abstraction of scene modeling, to integrate the building of geometric forms with the unique features of IBP, such as ports and interaction constraints. The action graph constructs IBP scripts for animating the constructs of the model graph. The scripts apply software interactions to bind the constructs of the model graph with computational components, such as keyframe interpolators. The scripts ultimately decide how the visual simulations update state and vary over time.
graph produces meta-scripts for controlling the temporal properties of the action graph. The action graph indicates what happens in a scene while the time graph indicates when the actions of the scene occur.

The diagrams of Figure 7.1 exemplify the basic designs of the three graphs. Each graph appears as a collection of nodes, which are either structural or descriptive. Structural nodes organize the topology of the graph while the descriptive nodes modify and establish state. The nodes of each graph vary in design and application. The nodes of the model graph form geometric shapes with location, orientation, and color. The ordering of nodes establishes a distinctive association between geometries and material attributes. The nodes of the action graph organize software interactions into distinct groups. Each group applies a separate logic to the ordering and execution of software interactions. The nodes of the time graph specifies a temporal ordering to scripts. Each node of the time graph inherits an interval of time that begins at the root and changes continuously as it travels down to the leaves.

![Diagram](a) Model Graph (b) Action Graph (c) Time Graph

Figure 7.1: The three graphs of the general specification

The remainder of this chapter details the definition of the general specification with its three graphs and the use of the general specification in developing a general file format for animation. The descriptions of the three graphs include a discussion of their parts and an evaluation of their application. The utility of the three graphs is demonstrated through the creation of the animation of the general example. The chapter concludes with a discussion of the relationships between the general specification and two popular specifications for scene modeling, VRML and JAVA3D.

### 7.1 Model Graph

A model graph is an acyclic, hierarchical structure for assembling geometric forms. The hierarchy of nodes within a model graph defines a modeling state, a set of transformations
and visual properties that effect the size, material attributes, and orientation of geometrical
primitives. The positioning of nodes in a model graph decides the value of the modeling
state and the application of the modeling state to the geometrical primitives.

### 7.1.1 Parts

As shown in Table 7.1, the nodes of the model graph are one of four kinds: *shape*, *transform*,
*appearance*, and *group*. The first three appear as leaves of a model graph while the last
establishes the topology of the graph and forms relations between nodes. Shape nodes
represent geometrical primitives, such as spheres, cubes, and nurb surfaces. Transform
nodes apply geometrical transformations, such as translation, scaling, and rotation, to the
shape nodes so as to decide their position, orientation, and size. Appearance nodes assign
material attributes, such as color, opacity, and reflectivity, to the shape nodes. Group nodes
unite shape nodes, transform nodes, and appearance nodes into a single assembly. The
composition of an assembly determines the associations between nodes and the scope of
each node's influence within the hierarchy. *GEOVIEWSWITCH*, for instance, limits the scope
of its nodes according to the viewing conditions of a virtual camera.

<table>
<thead>
<tr>
<th>Type</th>
<th>Kinds</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHAPE</td>
<td>GeoSphere, GeoCube, GeoCylinder, GeoSpline, etc.</td>
</tr>
<tr>
<td>TRANSFORM</td>
<td>MatrixRS, QuartMatrix, etc.</td>
</tr>
<tr>
<td>APPEARANCE</td>
<td>Material, TextureMap, Complexity, LightingModel, etc.</td>
</tr>
<tr>
<td>GROUP</td>
<td>GeoHorzComp, GeoSwitch, GeoViewSwitch, etc.</td>
</tr>
</tbody>
</table>

Table 7.1: Kinds of model graph nodes

### Fields and Ports

Every node in a model graph consists of a collection of *fields* and *ports*. The fields are
identifiers for storing the state of a node and for establishing the node's state space. A
cylinder, for instance, usually contains two fields, one for its radius and another for its
height. Ports are first-class, unidirectional operations that manipulate the values of fields.
As discussed in section 4.2.2, ports freely support software interactions and the methods of
IBP. For instance, the ports *inSetRadius* and *outRadius* are two ports affecting the value
of a cylinder's radius. In conjunction with software interactions, the first port imports a new
value for the radius while the second port exports the radius' current value.
7.1.2 Evaluation

The evaluation of a model graph begins with a default state and progresses depth-first from the root of the graph to individual leaves. The default state, which consists of the identity matrix and a list of visual properties, defines the initial values of the modeling state. As the evaluation of a model graph progresses, the modeling state updates according to the placement and definition of individual nodes. The nodes of type TRANSFORM update the modeling state with a transformation matrix while the nodes of type APPEARANCE update the modeling state with visual properties.

The group nodes of a model graph decide how changes to the modeling state accumulate and how branch points in the model graph distribute their inherited modeling state to their descendents. Changes to the modeling state are normally local to a group node and affect only those nodes that are within the same group or further down the hierarchy. For instance, the material node of Figure 7.1a creates changes to the modeling state that affect only the sphere node, which resides in the same group node as the material node. The cube node is unaffected because the cube node is neither a member nor a descendant of the group containing the material node.

The shape nodes of the graph, such as the cube, inherit the modeling state set by their positions in the hierarchy. Shape nodes normally inherit only those changes to the modeling state that are set by their predecessors. Thus, a shape node near the root of a graph inherits a modeling state quite different from that applied to a shape node near the leaves of the graph.

7.1.3 File Format

The composition and representation of the model graph readily supports a device-independent file format. As shown in Figure 7.2a, the definition of a node consists of a name, a nodeType, and a list of fields and ports. The name of a node is optional and required only when other nodes, such as those of the action graph, reference the node by its name.\(^1\) The nodeType of a node is an identifier of the node's type, which is any of a variety of kinds, as presented in Table 7.1. All fields and ports, as listed within a node, are declared with three identifiers. A field consists of a dataType, a fieldId, and a fieldValue. The dataType indicates a type, such as a float or integer; a fieldId provides a name; and the fieldValue presents an initial value. A port consists of a portType, a portId, and a dataType.

\(^1\)As in VRML, the name of a node is preceded by the term DEF and a reference to the same node by its name is preceded by the term USE.
The portType identifies a port's type, which always indicates a port's direction, while the portId and dataType provide the port a name and declare what kind of information the port communicates.

```
[ DEF <name> ] <nodeType> {
  [ <dataType> <fieldId> <fieldValue> ]
  [ <portType> <portId> <dataType> ]
  { body }
}
```

-A-

```
DEF body GEO-CYLINDER {
  RS_INT radius 10.0
  OPORT outRadius RS_FLOAT
  IPORT inRadius RS_FLOAT
}
```

-B-

Figure 7.2: The syntax of a node from the model graph

The text in Figure 7.2b presents a definition of the node body as it would appear in a data file. Of type GEOCYLINDER, body declares one field and two ports, which are a subset of its complete list of parts. The field radius stores a floating point number, while the ports outRadius and inRadius each communicate a floating-point number. The first port emits a number for a software interaction to read while the second port accepts a number for a software interaction to write. The parts that are missing from body, such as its height, are given default values as set by the node's definition.

### 7.2 Action Graph

Similar to a model graph, an action graph is a hierarchical structure. The hierarchy of the model graph assembles geometric forms while the hierarchy of the action graph assembles IBP scripts. Unlike the model graph, which produces a single structure, the action graph consists of four separate parts, all working together to coordinate the sequencing and execution of software interactions. The first part defines nodes that are local to the definition of a script. The remaining three parts apply a hierarchy of nodes to develop a data structure that either manages or simplifies the building of IBP scripts with software interactions.

#### 7.2.1 Parts

The four parts of an action graph are LOCALS, EVENTS, VIRTUALS, and ACTIVITIES. The first three parts, which are all optional, simplify the development of the last part, which builds a script from a hierarchy of nodes. The LOCALS define individual nodes that are local
to the definition of the action graph. The **LOCALS** encourage the encapsulation of nodes within the body of scripts that apply them. The nodes appearing in the **LOCALS** are usually time-varying nodes, such as the hand and positional controllers of the general example, or helper components, such as the color controller of the extended example. These nodes have limited scope and are best introduced within the context of the action graph.

The **EVENTS** define the software interactions of the script that are commonly applied by its **ACTIVITIES**. These interactions communicate information between either the nodes of model graphs or the nodes appearing in the **LOCALS**. Hierarchical structures develop within the **EVENTS** as individual interactions use or organize additional interactions. For example, a conditional interaction often applies other interactions to link the communications of a system to run-time conditions.

The **VIRTUALS** define interaction constraints for the script to apply in simplifying the definition of complex communications. The constraints either bind the ports of model graphs, which may appear in the **LOCALS**, or link the execution of software interactions, which appear locally or in the **EVENTS**. The **VIRTUALS** develop hierarchically as individual constraints repetitively apply other constraints in their definitions. The code in Figure 5.2, for instance, introduces a chain of interaction constraints. The chain progressively constructs a hierarchy of nodes as it accesses additional numbers of ports and interaction constraints.

The **ACTIVITIES** apply the first three parts in producing an IBP script from an ordering of nodes, of which there are three kinds: **interactional**, **operator**, and **grouping**. Interactional nodes define software interactions. The nodes define the interactions explicitly or by referencing the interactions defined in the **EVENTS**. Normally, software interactions that appear once in the hierarchy manifest in the **ACTIVITIES** while those that appear multiple times or those involved in interaction constraints manifest in the **EVENTS**. Operator nodes define an ordering, such as parallelism or sequence, to interactional nodes. Grouping nodes assemble interactional nodes into groups, with each group applying an alternative method to order its nodes. The groups order their nodes with either functional or declarative methods, as discussed in section 6.2.1, or a combination of both. The hierarchy of the action graph grows as the **ACTIVITIES** repetitively embed nodes into groups and as interactional nodes reference other nodes in their definitions.

---

2In general, the model graph and its nodes are globally visible to all other nodes, from either the action graph or time graph. To support greater encapsulation of information, model graphs may be defined locally within an action graph.
7.2.2 Evaluation

The evaluation of an action graph begins with an examination of its LOCALS and follows with an evaluation of its VIRTUALS, EVENTS, and ACTIVITIES. The LOCALS always appear first while the three remaining parts appear or repeat in any order that maintains consistency amongst its nodes. An ordering of parts is consistent when each of its parts references only the nodes of other parts that appear earlier. For instance, if an ACTIVITIES accesses the interaction of a particular EVENTS, that EVENTS must appear before the definition of the ACTIVITIES. Otherwise, the ACTIVITIES will attempt to access an interaction that has not yet been defined.

The evaluation of the LOCALS is similar to that of the model graph, but unlike that of the other parts, which construct data structures. The LOCALS apply nodes in developing and updating a modeling state while the VIRTUALS, EVENTS, and ACTIVITIES apply nodes in constructing data structures for managing software interactions. The data structures arise with the evaluation of the hierarchy, which begins at the top of a hierarchy and progressively visits each level depth-first. Each level of the hierarchy either provides further information to the level above, as occurs with VIRTUALS and EVENTS, or presents greater information regarding the relations between nodes, as occurs with the ACTIVITIES.

7.2.3 File Format

The composition and representation of the action graph decompose readily into a device-independent file format that shares commonalities with the file format of the model graph. As shown in Figure 7.3, the definition of an action graph consists of a name, an actionType, a list of optional parts, and a collection of ACTIVITIES. The name of an action graph is purely informative and is required only when the action graph is referenced by its name. The actionType of an action graph is a label for distinguishing a script by its type. Currently, the only available type is “Script,” which denotes the assembling of software interactions into distinct ACTIVITIES.3 The optional parts appear in sequence, with each part defined by a title and a list of nodes, arranged sequentially or in a hierarchy. The title identifies the list of nodes as LOCALS, EVENTS, or VIRTUALS. The nodes of the ACTIVITIES create an ordering for software interactions while arranging the interactions into individual threads.

The nodes associated with the LOCALS appear in the same format as those of the model graph (section 7.1.3). Every node appears with a name, a type, and a listing of ports and fields. The nodes associated with the EVENTS present software interactions. As

---

3New types will undoubtedly become available as IBS improves in functionality and design.
found in Figure 7.4a, each interaction appears with a name, an *interType*, a list of fields. The name, set by the same naming convention as that of the model graph, is a required identifier. The *ACTIVITIES* of the action graph reference a node by its name when arranging the nodes into a sequence. The *interType* identifies a node’s type and the interaction that the node represents. As discussed in section 4.2.2, software interactions are of many types, and each references an assortment of ports and filters. All fields are declared with two identifiers: *fieldId* and *fieldValue*. The *fieldId* provides a name and the *fieldValue* presents an initial value, which may be any numerical constants or the names of nodes or ports, as set by the naming convention established by the model graph.

The text of Figure 7.4b, for instance, presents the definition of *EventA*, a conditional interaction containing a source and body. The source references the port *OP_BOOLEAN* from the node *ID_COMP* while the body defines *EventB*, a second interaction containing a source, target, and filter. *EventB* communicates information between the ports *OP_POS3* and *OP_POS2* of the nodes *ID_COMP1* and *ID_COMP2* via the filter *ID_FILT*. The second interaction is local to the definition of *EventA* and executes each time that *OP_BOOLEAN* provides a positive value.
The nodes associated with the virtuals define interaction constraints. As shown in Figure 7.5, the interaction constraints apply the same naming convention as that applied by the event in referencing ports and interactions. Every interaction constraint appears as an expression containing either an operator or a function. Both operators and functions accept references to ports, virtual ports, or numerical constants. The name of the interaction constraint is set by the return value of its expression. The example in Figure 7.5 presents two interaction constraints, VP_HAND and VP_CONTACT. The first constraint sums the value of two ports while the second computes the distance between the first constraint and the origin.

\[
\begin{align*}
   \text{VP_HAND} &= [\text{ID.ROB, OP-HAND}] + [\text{ID.ROB, OP.POS}] \\
   \text{VP_CONTACT} &= \text{distance}(\text{VP_HAND, ORIGIN})
\end{align*}
\]

Figure 7.5: The syntax of the virtuals of an action graph

The nodes associated with the activities assemble an ordering to the software interactions of the script. As appearing in Figure 7.6a, every node appears with an association, an orderingType, and a collection of threads. The association links the node to an identifier, such as “InitialEvents” or “ActingEvents,” that determines the node’s role in a script. The association “InitialEvents,” for instance, is understood by a script to execute immediately after the script begins. The identifiers understood by a script vary according to the script’s type. The orderingType of a node specifies a particular ordering to a set of interactions. Different types apply alternative methods to decide the frequency and order of software interactions.

The threads assemble software interactions into lists. Each list begins with an optional label containing a numerical value, which by default is zero. Threads apply operators to the list of interactions, which are defined either in the events or locally within the list, to identify a distinctive ordering, such as that involving parallelism or repetition. Figure 7.6b, for example, creates a single thread with four interactions, of which the last is defined locally. The four interactions occur in sequence with the second and third interactions occurring in parallel, as indicated by the + operator.
### 7.2.4 Extended Example Revisited

The diagram in Figure 7.7 visualizes the action graph of the script appearing in Figure 5.4.\(^4\) The graph contains eighteen nodes, of which two are grouping, thirteen are interactional, and two are operator. The grouping nodes partition the interactions into two separate sequences: INITIAL EVENTS and ACTING EVENTS. The interactions of the former execute immediately after the script begins while the interactions of the latter execute repeatedly whenever the script is active. The two operator nodes, both of kind +, execute interactional nodes in parallel. The first operator executes evA with evB while the second executes evE with evF.

![Activity Diagram](image)

Figure 7.7: The action graph of Figure 4.4

The text in Figure 7.8 presents a listing of the action graph of Figure 7.7. Entitled "lightbyRobot," the listing appears as it would in a file or database for sharing and storing the script. The listing begins with the optional parts of the script (lines 2-31) and concludes with the ordering of two lists of interactions (lines 32-40). The operational parts define a filter, three interaction constraints, and sixteen software interactions. The filter is applied by

\(^4\)The code of Figure 7.7 accesses several global variables, which are not shown for the sake of brevity.
ID.evF to connect two components that communicate different kinds of data. The interaction constraints create virtual ports for determining whether a robot selects a button. The listing concludes with two lists of interactions, established by the nodes InitialEvents and ActingEvents (lines 32-37). Each node specifies a different sequence for a separate selection of interactions.

7.3 Time Graph

A time graph is a hierarchical data structure that describes the temporal characteristics of an animated scene. As illustrated in Figure 7.9, it consists of a hierarchy of nodes for building a temporal state and assigning the state to scripts (or even other time graphs). The temporal state establishes a set of values and conditions that decide when and how scripts operate. A time graph of greater size and depth produces numerous changes to the temporal state and generally describes scenes of greater complexity.

A time graph is similar to a model graph in that the positioning of its nodes is instrumental in determining how its nodes relate to each other and how its nodes inherit the graph's state. In a time graph, script nodes inherit the temporal state established by their positions. The two graphs are dissimilar in that the time graph permits cycles while the model graph does not. A cycle in a time graph is useful for producing repetitive behaviors and re-occurring events.

7.3.1 Parts

As shown in Table 7.2, there are five types of nodes of a time graph: timers, relations, modifiers, groups, and scripts. Timers, modifiers, and scripts form the leaves of a time graph while relations and groups organize the leaves into a hierarchy. Timer nodes specify the temporal characteristics of a scene by manipulating an interval of time, controlling temporal properties, and establishing temporal delays. An interval of time defines a range of time upon which a script bases its computation. From the interval, the computations obtain a duration and time step. Some nodes, such as TMINTEVAL, initialize an interval's range while others, such as TMSHIFTINTERVAL, apply transformations to an interval to shift or scale its range. Temporal properties control the attributes of an interval, such as granularity. An interval with high granularity causes a script to update its computations frequently. Temporal delays relate the passage of time. They extend an interval of time to include a period of inactivity upon which scripts cease to work.

Relation nodes establish temporal relations among intervals of time. Appearing in
Table 7.3, the temporal relations establish time-based conditions for deciding when an interval begins and ends. Some relation nodes, such as TmDELMITS, initiate and terminate intervals simultaneously while others, such as TmMEETS and TmSEQUENCE, order intervals sequentially or according to the advancement of time. With relation nodes, the execution of
Type | Kinds | Purpose
--- | --- | ---
**TIMER** | TmInterval; TmScaleInterval; TmShiftInterval; TmNextInterval | establishes and alters interval
 | TmGranularity; TmScaleGranularity | establishes and alters temporal update
 | TmHaltTime; TmSuspendTime; TmWaitTime; | stochastic controls
**RELATION** | TmStarts; ReStarts; TmMeets; ReMeets; TmDelimits; ReDelimits; TmStops; TmSequence | applies temporal relations
**MODIFIER** | TmCondition; TmReActivate; TmInactive; TmCounter | modify state info of time graph
**GROUP** | TmGroup; TmSwitch; TmToggle | collates nodes

Table 7.2: Kinds of time graph nodes

A script need not always begin and end at preset times. One script may begin after another one ends.

<table>
<thead>
<tr>
<th>α Relation β</th>
<th>Description</th>
<th>Inactive β</th>
<th>Active β</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TMSTOPS</strong></td>
<td>when α starts, β stops</td>
<td>N/A</td>
<td>stops</td>
</tr>
<tr>
<td><strong>TMSTARTS</strong></td>
<td>when α starts, β starts</td>
<td>starts</td>
<td>restarts</td>
</tr>
<tr>
<td><strong>RESTARTS</strong></td>
<td>when α starts, active β stops and restarts</td>
<td>N/A</td>
<td>restarts</td>
</tr>
<tr>
<td><strong>INSTARTS</strong></td>
<td>when α starts, inactive β starts</td>
<td>starts</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>TMDELIMITS</strong></td>
<td>when α starts (stops), β starts (stops)</td>
<td>starts &amp; stops</td>
<td>starts &amp; stops</td>
</tr>
<tr>
<td><strong>REDELIMITS</strong></td>
<td>when α starts (stops), β restarts (stops)</td>
<td>N/A</td>
<td>starts &amp; stops</td>
</tr>
<tr>
<td><strong>INDELIMITS</strong></td>
<td>when α starts (stops), inactive β starts (stops)</td>
<td>starts &amp; stops</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>TMMEETS</strong></td>
<td>when α stops, β starts</td>
<td>starts</td>
<td>restarts</td>
</tr>
<tr>
<td><strong>REMEETS</strong></td>
<td>when α stops, active β stops and restarts</td>
<td>N/A</td>
<td>restarts</td>
</tr>
<tr>
<td><strong>INMEETS</strong></td>
<td>when α stops, inactive β starts</td>
<td>starts</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 7.3: The effects of temporal relations on active and inactive scripts

Modifier nodes assign conditions to the temporal state and their application to scripts. The conditions relate the temporal state to special occurrences, such as the states of run-time variables or the past execution history of a script. For example, the modifier node **TMINACTIVE** indicates that temporal state affects only scripts that are inactive. When applied with **TMDELIMITS**, the **TMINACTIVE** node produces **INDELIMITS**, a temporal relation behaving similarly to **TMDELIMITS** but working only with scripts that are inactive.

Group nodes arrange the nodes of the time graph into groups, with each group inheriting a single interval of time. The group nodes decide which nodes of a group are active and how the nodes of a group relate. For instance, the nodes **TMGROUP** and **TMSWITCH** apply different means to manage a set of nodes. The former group node activates the entire set while the latter activates only one at a time.
7.3.2 Evaluation

The evaluation of a time graph involves a depth-first traversal of its hierarchical structure. The traversal process begins with default values and works incrementally to build a temporal state and apply it to every script in the graph. Changes to the temporal state evolve as the traversal process encounters nodes of varying types, as shown in Table 7.2. Each new node produces a new change or modifies a previous update. Nodes along the same path from root to leaf produce incremental changes while those along differing paths produce varying results. As the traversal process returns to a previous level, it undoes the changes performed by nodes of lower levels. For example, upon returning from the TMMEET node of Figure 7.9, the changes of the TMREACTIVATE node are undone. The effects of the TMREACTIVATE node apply only to script A, not to scripts Z or E.

The top node of a time graph is normally an TMINTERVAL node. The node establishes an initial reference interval for the remainder of the tree to inherit. An interval is defined as a span of time with a beginning and an end. Every non-TMINTERVAL node of the hierarchy forms its own interval by inheriting the interval of its parent. They either copy the interval or derive a new one by applying modifications or invoking a temporal relation. Every script

---

5The beginning and end of a interval may coincide to represent an instantaneous moment in time.
within the hierarchy also inherits the interval of its parent. The inherited interval determines
the times that the script begins and ends.

The illustration and code segment of Figure 7.9 presents a sample time graph with
each of the three node types of Table 7.2. The TMINTERVAL node establishes an interval
of time for the TMSTARTS node to inherit. The TMSTARTS node instructs script A and the
TMSEQUENCE node to begin simultaneously. The TMSEQUENCE node sequences through
scripts R, Z, and E. The TMGRANULARITY nodes modifies the update frequencies of scripts
Z and E. The TMMEETS node, with the modifier TMREACTIVATES, indicates that script A
"restarts" when script R finishes.

7.3.3 File Format

Like both the model graph and action graph, the time graph decomposes readily into a
device-independent file format. As shown in Figure 7.10a, the definition of every node
within a time graph consists of an optional name, a timeType, and a body. The timeType
identifies the node as a type of timer, relation, modifier, or group. The body consists of
either fields with values or references to nodes and scripts. The fields set the state of a
node while the nodes and scripts provide the node with data. The listing of Figure 7.10b,
for example, defines movements, a group node containing an TMINTERVAL node and two
scripts. The TMINTERVAL node defines a span of time for the two scripts to reference in
their computations.

```
[ DEF <name> ] <timeType> { 
[ <dataType> <fieldId> <fieldValue> ]
[ <node> | USE <script> ]
} 
-A-
```

```
DEF movements TmGroup {
  TmInterval {
    RS.FLOAT begin 20.0
    RS.FLOAT end 30.0
  },
  USE ID_moveBall, USE ID_moveHand
} 
-B-
```

Figure 7.10: The syntax of a time graph

7.3.4 Contextual Example Revisited

The illustration and code segment of Figure 7.11 present the time graph for the large scene
presented in Figure 3.3. The time graph consists of fifteen nodes, of which five are timers,
three are temporal relations, and two are groups. The five remaining nodes reference four scripts: robLight, robPush, mvBox, and mvBarge. The two references to robLight apply two temporal relations to the script, one for delimiting and another for stopping.

```c
/* robot lights stationary light */
Script *robLight = RobotLight(...);
/* robotB pushes boxes */
Script *robPush = RobotPush(...);
/* cranes moves boxes */
Script *mvBox = moveCrane(...);
/* barge moves */
Script *mvBarge = moveBarge(...);
```

![Diagram of a time graph](image)

Figure 7.11: A time graph of the extended example

The root of the time graph initializes the program with a reference interval. The interval begins at ten and terminates at infinity. The nodes of level one and level two indicate that the execution of robLight delimits the execution of levels three and four.
The two lower levels begin and end with the activity of robLight. The nodes of level three indicate that robPath starts the actions of mvBox and mvBarge. mvBox starts simultaneously with robPath while mvBarge commences after a short delay. The nodes of level four alter the temporal state of robPath and mvBox, and relate mvBarge to robLight. The temporal states of the two scripts differ either by their granularity or by their halting time. The actions of mvBarge terminate the actions of robLight and eventually, the entire program. When mvBarge stops, robLight terminates the scripts of levels three and four.

The text of Figure 7.12 presents a listing of the time graph of Figure 7.11. The listing embeds multiple node definitions into a single hierarchy containing eleven nodes. All of the nodes appear locally within the hierarchy while the definitions of the scripts appear elsewhere.

```plaintext
(1) DEF doLight TIME-GRAPH {
(2)   TmGroup {
(3)     TmInterval {
(4)       RS.FLOAT begin 0
(5)       RS.FLOAT end infinity
(6)     }
(7)     TmDelimits {
(8)       USE ID.robLight,
(9)       TmHaltTime { ... },
(10)      TmStart {
(11)        TmGroup {
(12)          TmGranularity { ... } 
(13)          USE ID.robPush,
(14)        }
(15)        TmGroup {
(16)          TmHaltTime { ... },
(17)          USE ID.mvBox,
(18)        }
(19)        TmShiftInterval { ... },
(20)        TmStops {
(21)          USE ID.mvBarge,
(22)          USE ID.robLight
(23)        }
(24)        }
(25)      }
(26)   }
(27) }
```

Figure 7.12: The file format for the time graph of Figure 7.7
CHAPTER 7. GENERAL SPECIFICATION

7.4 Comparison to VRML and JAVA3D

The general specification and existing specifications for scene modeling, in particular VRML and JAVA3D, are similar yet noticeably different. They share several similarities in their usage of hierarchies but contrast dramatically in their approach to developing and evaluating the hierarchies.

7.4.1 Similarities

Both the general specification and existing specifications for scene modeling rely upon a hierarchical graph for developing software from an integration of parts. The arrangement of parts in a graph defines an ordering to the parts and ultimately decides what the parts do as a whole. The general specification applies the graph to the development of animation, while VRML and JAVA3D apply the graph to the modeling and rendering of geometry. The graph is an appropriate structure because it is simple to build, easy to evaluate, and quick to express as a device-independent file format. Altering the composition or structure of a graph is a simple way to produce new and interesting results.

Applications that apply either of the specifications benefit by exporting and disseminating their results. The applications store their results in a general format that is easy to interpret and readily understood. Applications reproduce the results of others by organizing their parts into similar graphs. The applications need neither link continuously with new code nor accommodate the designs of others.

7.4.2 Differences

The differences between the general specification and existing specifications for scene modeling are far more significant than simply a difference in goals or the number of graphs. Each of the three graphs of the general specification differs from the scene graph either in the design of their parts or in the methods they apply in evaluating their nodes.

Model Graph

The model graph and scene graph share similarities in node types but differ in their means of supporting communications. The nodes of the model graph extend the nodes of the scene graph to use ports rather than fields when animating state. Fields, as specified by VRML, are first-class variables for storing state information. When involved with communications, the fields simply return or update their own state. Ports operate dissimilarly in that they
react to communications. They set the states of fields by either executing computational
commands or interfacing with the software interactions of IBP. The ports either apply
algorithms in updating the values of fields or work with interactions in supporting greater
forms of communications, such as partial writes (see section 8.1.1).

With ports complementing fields, the nodes of the model graph offer several advan­
tages over the nodes of the scene graph. These advantages include an increase in functional­
ity and a gain in support for software interactions. The functionality of a node increases as
it is able to react to its communications. By executing commands or algorithms, the ports
of nodes behave as methods, rather than data members. Unlike fields, the ports may reject,
accept, or modify data as the data enters or leaves a node. For instance, one of the ports
serving the robot of the general example not only receives a position for the robot’s hand
but also configures the angles of the robot’s joints. Had the robot simply accepted a position
for the hand and not set the values of its joints, the joints would not accurately reflect the
robot’s state.

Action Graph

Unlike the scene graph, the action graph provides direct support for organizing the commu­
ications of a system. The scene graph supports limited means for animating the topology
of its interconnections and the states of its nodes. To animate the state of the scene graph,
VRML applies routes, simple interconnecting mechanisms for communicating state changes
between fields. When state changes occur, the routes move information from one field to
the next. The use of routes and is performed haphazardly or with the use of scripting nodes,
which encapsulate the instructions of a scripting language. Neither approach is as useful or
as easy to apply as the general specification. Haphazard use resolves only short-term gains
while scripting nodes merely encapsulate code and support few aids in animating state.

Normally, the scene graph and its nodes are globally visible to all other nodes. This
is unlike the general specification which extends the rule of scoping to its definitions. The
nodes of any graph may appear globally or as locals within another. For instance, the
LOCALS of the action graph encourage the encapsulation of time-varying nodes within the
scripts that apply them. In constrast, VRML distributes the definition of its time-varying
nodes, referred to as time-dependent nodes and interpolator nodes, globally throughout its
body.

Lutterman and Grauner [92] proposed an alternative approach, based upon the use of various
"history" nodes, to vary the states of an scene graph over time. These nodes are best for presenting
spatio-temporal data, not for developing a basis for animation.
Time Graph

The evaluation of a time graph is similar to that of a scene graph (VRML, JAVA3D, etc.). Both involve the incremental building of state information with a hierarchical structure and each evaluates its respective graph with a complete depth-first traversal. However, the two graphs apply a different approach in evaluating their nodes. Whereas the evaluation of a scene graph is usually immediate, the evaluation of a time graph is not. Most changes described by a scene graph are not linked to run-time conditions. Except for nodes that link to scripts or interactive widgets, the final state and appearance of a scene graph are readily determinable. For a time graph, the results of an evaluation are not known until a program runs and simulation time advances. The run-time conditions of a program are very important for deciding the final outcome of the time graph's description. To advance simulation time simultaneously with the evaluation of the time graph would be impractical and unwise. Such a task would require that the evaluation of several paths through the tree occur in parallel. Many nodes, such as START and DELIMIT, require simultaneous execution of multiple scripts. For scripts along different paths, such as A and R appearing in Figure 7.9, it is important that each is processed to start together. If time advances between the evaluation and execution of each, the program produces erroneous results.
Chapter 8

Reuse Guidelines

The structuring of a program into a system of software components interconnected with software interactions does not guarantee the reusability of an interaction-based program or the reliability of its parts. It is easy to develop software components that work poorly in an interaction-based environment. The components may selfishly hard-code or initiate personal communications, or ignore the requests of software interactions in sending and receiving information. Hence, the reuse of an interaction-based program requires guidelines to govern the development of software components. The rules standardize the development and application of code so as to accommodate the workings of software interactions and to support the use of software components in an interaction-based environment. As noted by professionals and researchers, successful reuse requires that software components be developed with the intent of being reused [19, 60, 71].

Current proposals for component properties, such as correctness, efficiency, simplicity [66], composability [17], certifiability [162], and self-restraint [101], are helpful in developing components for IBP, but insufficient for ensuring that the components support IBP. In an interaction-based environment, software components experience dynamic “plug-and-play.” As software interactions execute, software components “plug” into a software network and “play” to communicate information. During a program’s execution, the software components participate in a large number of networks of varying configurations and designs. Within each network, the software components undergo new roles and provide new services. Unfortunately, current proposals do not recognize the ramifications of producing components for a plug-and-play environment. For IBP to operate properly, software components must not only behave uniformly, but must also support the interactions that control them.

This chapter introduces a new set of properties, of which there are three groups, to complement the current proposals for component design [89]. These properties aid in
developing software components that work well in interaction-based environments and reuse easily in multiple contexts. The three groups of properties are interactional, behavioral, and temporal. Interactional properties determine how components work with interactions and communicate information. Behavioral properties determine how components respond to communications and complete their computational tasks. Temporal properties determine when components act and how components integrate their computations with time.

This chapter describes the three groups of properties and explains the relative attractiveness of each. Not every property applies equally well to every component. To promote extensive reuse, components should accommodate as many properties as possible. Components that apply non-conforming properties benefit from short-term gains, but suffer in not reaping the benefits of IBP. The description of each group is accompanied with a detailed examination of the components of the contextual example. In supporting the three groups of properties, the components ready themselves for greater reuse.

8.1 Interactional

Interactional properties determine how components work with interactions and how components communicate information. The properties ensure that software components permit software interactions to control freely all forms of communications, and that software components provide a general interface consisting of unidirectional, first-class operations. With free-flowing communications, software interactions decide solely on how and when components communicate. With first-class operations, software interactions initiate queries to manipulate and to verify communicative tasks.

8.1.1 Properties

Property I–1: Components should permit interactions to control communications freely.

Interactions freely control communications if they, not components, determine when and how information flows. Components should wait for interactions either to provide or to solicit information. If necessary, components may inform interactions when they are ready to communicate, but they should never prevent a communication from occurring or order a communication to occur. When components wait (during push requests\(^1\) and callbacks) for interactions to further their state, they ignore newly incoming information and remit old information as an outgoing substitute.

\(^1\)A push request involves asking a recipient to receive and to handle an outflow of data.
By permitting interactions to act freely, components may assume a wide variety of roles and perform in many contexts. A component asserting communicational restrictions, such as that of requiring an immediate transfer of information, is useful only in contexts in which an interaction continuously monitors the component’s state. In any other context, guarantees are unavailable to ensure that the transfer occurs, let alone that the transfer occur immediately. For visual simulation and other dynamic systems, communicational restrictions entail grave consequences: they limit the types of topological changes that may occur and the type of systems that may be described.

**Property I-2:** Components should provide first-class operations.

First-class component operations are essential in interaction-based programming as they permit an interaction to query an operation about its state prior to its use. As discussed by Lee [87], not all interactions operate instantaneously or in fixed sequence. Many, such as those employed in dynamic systems, remain inactive until important times or states arise.

An interaction queries an operation for its state to determine the datatypes supported by the operation and to ascertain the identity of the most recent interaction from which an operation communicated. The former query ensures proper communication between components while the latter query prevents simultaneous write access, the act of updating of a single state variable with multiple values. For time-varying systems, such as visual simulation, simultaneous write access is a problem warranting careful attention. As time advances, the value of a state variable may update only once per time-step, since all state changes are instantaneous. State changes occurring simultaneously introduce errors since they are unpredictable and frequently misinterpreted.

First-class component operations also provide benefits to software components. Specifically, components gain the ability to monitor and register carefully the activity of strongly related operations, those which update similar state variables. Components may either register the operations manually or have the operations register themselves automatically. For example, a component may register two operations to be strongly related so as to signal a write access fault when both execute simultaneously. Such a situation would occur if two operations, acting simultaneously, apply different means in setting the base-joint angle of an articulated figure. The first operation sets the angle directly while the second operation sets it indirectly by setting the position of the robot’s hand.

**Property I-3:** Components should provide a general interface, one where each operation is either a unidirectional read, a unidirectional write, or a bidirectional write.
An interface is general if it provides a consistent means to read and write information, and if it permits interactions to act freely. An interface supports these traits if its operations are unidirectional read, unidirectional write, or bidirectional write (see Figure 8.1). A unidirectional read operation simply supplies an interaction with data and thus operates without input parameters. A unidirectional write operation acts similarly, except that data flows inversely. In this case, an interaction passes data to the operation. A bidirectional write operation permits an interaction to obtain information from a component before the interaction writes data to the component. This operation behaves in a manner opposite to that of a bidirectional read, which is commonly called a function and is not a part of a general interface.

![Figure 8.1: Interface methods for communicating information](image)

Unidirectional reads and unidirectional writes are beneficial because they permit interactions to control freely the movement of data. The interactions read and write data to the operations when appropriate. Bidirectional write operations are beneficial in that they simplify the coding of two communicative operations, read and write, into one. In addition, they support the implementation of partial writes: the passing of incomplete information to write operations that accept aggregate datatypes. Operations that support partial writes accept incomplete information and produce fewer changes than those that receive a whole datatype, complete with information. For example, operations that modify an aggregate \(x,y\) pair support partial writes if they, upon receiving only one value, modify either \(x\) or \(y\), but not both. The missing value neither introduces changes nor receives a default value.\(^2\)

Partial writes increase the reusability of components by expanding their capabilities and by simplifying their interfaces. Operations that accept partial writes displace all operations that apply equivalent changes to only a subset of the fields comprising the aggregate

---

\(^2\)Only a special class of operations accept partial writes. These operations read and write aggregate datatypes, and handle each of the fields of the datatypes separately.
datatype. In other words, a single operation that is variadic may replace many individual operations, which may or may not address every possible combination of field values.

**Corollary I–3:** Components should avoid the use of bidirectional read operations.

Acting in a manner opposite to that of bidirectional write operations, bidirectional read operations require an interaction to provide them with information before the interaction is provided with data. For interaction-based programming, bidirectional read operations lead to two major problems. First, such operations compel interactions to communicate the results of initial write operations immediately, undermining the authority of interactions to choose when and how communications occur. Second, such operations hinder multiple interactions from observing the results of the initial write operations. To reproduce the original results, additional interactions must either supply the operations with duplicate arguments or inform the operations to return their most recent results. Either way, each approach introduces additional complications. The former requires interactions to share argument values, while the latter requires interactions to identify themselves as “primary” or “secondary” readers. Primary readers request new information while secondary readers query for information given to previous requests.

### 8.1.2 Contextual Example Revisited

It is important that the components of the contextual example support the interactional properties so that they can be effectively reused in multiple contexts, be it the context of the immediate program or of the context of another application. In observing Property I–1, which counsels that components should permit software interactions to control communications, the components are free from moderating their communications and may issue fewer restrictions on their use. As discussed in the contextual example, the use of software components, such as those controlling the motions of the robot, may vary widely between scenes. For example, in one scene, the components direct a robot to toggle a light, while in another, the same components manage a robot in moving a box.

When components rely on their own means to communicate, they complicate their reuse and their placement in software networks of differing topologies. For instance, if the robot of the contextual example communicates directly with a particular controller, other controllers encounter difficulties in applying new motions to the robot, such as that of placing boxes. To accommodate changing communications, the robot would either have to undergo continual modification or have to permit software interactions to introduce new
communications. The former is unfavorable in that it does not accommodate run-time changes, while the latter is undesirable in that it complicates the implementation of a robot.

In conformity with PROPERTY 1–2, which states that components should support first-class operations, the components of the contextual example enable software interactions to verify that data flows properly between components. For instance, both the positional controller and the robot respond to queries that ask for the types of data they communicate. Software interactions cross-check these types to ensure that both the controller and the robot communicate properly. In addition, the components are able to identify which of its operations are strongly related, those manipulating similar state variables. Accordingly, the robot may signal an error if it receives two notices to update its configuration or its position from two separate controllers in the same time-step.

With the use of strongly related operations, components aid in resolving programming errors that arise with conflicts in communications. In a time-varying system, such as that of the contextual example, those conflicts are difficult to unravel because the communications of the system occur at run-time and vary greatly with the use of stochastic techniques. Further enhancements to the contextual example may introduce additional actions to the robot, which in turn, create more opportunities for conflicts to develop. The strongly related operations are helpful in resolving these conflicts and in exploring the dynamics of a system, especially in learning how particular components respond to varying inputs.

In following PROPERTY 1–3 and its corollary, which requires that components support a general interface involving unidirectional operations, the components permit the writing of partial data, which would, for example, animate the red color channel of the light. The light is neither obligated to provide operations for animating individual channels, nor required to interact with a superfluous component that binds the individual channels of separate colors to produce a complete datatype. With unidirectional operations, the components promote fewer restrictions on their use. By not supporting bidirectional reads, the controllers of the contextual example, for instance, work with many kinds of interactions, not only those that are able to provide the controllers with a parameter value. With one providing a temporal input, several interactions may manipulate a single controller in updating the positions of several robots, such as those moving in single file. One interaction would write a value to the controller while the remainder would read and scale the output from the controller so as to stagger the positions of the robots.
8.2 Behavioral

Behavioral properties determine how components respond to communications and complete their computational tasks. These properties suggest that software components base their communication on their current state and that multi-parameter operations are partitioned into an elemental set of unary operations. The former ensures that software interactions transport valid information among components. It is not the responsibility of software interactions to verify that all components communicate their states accurately. The latter permits software interactions to communicate freely individual values to multi-parameter operations. The interactions, not the operations, decide the number of parameters that the operations send and receive.

8.2.1 Properties

**PROPERTY B—1:** *Components should always maintain and report a consistent state.*

A component should not expect or rely upon external stimuli to intentionally reconcile its state. Otherwise, the component may erroneously base its responses on state values that are either old or invalid. For instance, the diagram in Figure 8.2 presents an articulated figure and its corresponding component. The component computes the position of the figure’s elbow and hand from the joint angles, $\theta$ and $\phi$. The component acts in accordance with **PROPERTY B—1** if it re-computes the positions of the hand and elbow either immediately after it receives input, or immediately before it responds to future queries. An external stimuli is not required to re-compute the positions of the hand and elbow from past inputs.

![Figure 8.2: A component that obeys PROPERTY B—1](image)

It is important that a component continuously exhibit its current state for two reasons. First, it supports the separation of computation from coordination. It ensures that a component is not reliant on coordinational activities to reconcile its state. Second, it avoids potential race conditions that arise when multiple scripts interact concurrently with a single
CHAPTER 8. REUSE GUIDELINES

component. Difficulties arise if the execution order of the scripts intertwine improperly. For example, the illustration in Figure 8.3 presents a single component managed by two scripts, P and Q. The component accepts three inputs, with the names A, B, and K, and produces one output, C. The component computes C from A and B when notified by K. The script Q produces an inconsistent answer when Q-1 executes after P-2, but before P-3. Script Q computes a value of C that does properly reflect the values of A and B, as set by the first two lines of script P. Had the component been designed initially to conform to PROPERTY B-1, it would not need K and would always compute the correct value of C from its inputs, A and B. A component that follows this property ensures that the execution order of its communications is responsible only for determining when information is sent and received, not for deciding when the component advances its state.

\[ \begin{array}{c|c|c}
\text{A$_{in}$} & \text{C$_{out}$} & \text{Comp't} \\
\hline
\text{B$_{in}$} & & \\
\text{K$_{in}$} & & \\
\end{array} \]

Coordination Activities

\[ \begin{array}{c|c|c}
\text{P} & \text{Q} \\
\hline
\text{P-1: set A} & \text{Q-1: get C} \\
\text{P-2: set B} & \text{P-4: get C} \\
\text{P-3: set K} & \text{P-4: get C} \\
\end{array} \]

Figure 8.3: An inconsistent component with racing activities

**PROPERTY B–2:** Components should partition multi-parameter operations into an elementary set of unary operations.

The operations of a component are instrumental in establishing dependencies between the component and its environment. The environment provides the operations with parameter values that influence the computational state of the component. Operations that accept several parameters establish multiple dependencies, one for each parameter. An elementary set of unary operations partitions these dependencies into many individual ones. Each unary operation modifies one parameter value of the original operation. To complete the requirements of the original operation, the unary operations reuse the most recent values for the remaining parameters. For example, the operation \( \text{op}(A,B,C) \), which computes the function \( f(A,B,C) \), can be partitioned into three unary operations \( \text{opA}(A) \), \( \text{opB}(B) \), and \( \text{opC}(C) \). A call to any of these three operations updates one parameter and invokes \( f(A,B,C) \) with recent values for the two remaining parameters.\(^3\) Given the sequence "\( \text{opA}(X), \text{opB}(Y), \text{and opC}(Z) \)," the last call reproduces the effect of a single call to \( \text{op}(X,Y,Z) \).

\(^3\) Default values are stored for each parameter so as to handle cases in which the value of a missing parameter has never been set and thereby, has no recent value.
Partitioning multi-parameter operations into an elementary set simplifies the development of coordination activities. Interactions can freely decide when and how the components receive each of their parameter values. Partitioning also permits interactions to satisfy selectively a subset of the dependencies. The calling sequence of the unary operations determines the subsets. For example, the sequence \( \text{opB}(u), \text{opC}(v) \) updates two of the three parameters of the previous example. The sequence invokes \( f(A', u, C') \) followed by \( f(A', u, v) \), with \( A' \) and \( C' \) representing the last known values of \( A \) and \( C \). Finally, partitioning simplifies the interfaces of components. A component need not support a combinatorial number of operations to permit selective assignment. Interactions assign parameters selectively by invoking unary operations in sequence.

**Corollary B-2:** If a multi-parameter operation must produce side-effects, its corresponding set of unary operations should ensure that side-effects occur only once for a set sequence of parameter values.

Multi-parameter operations producing side-effects are troublesome because they do not partition easily into an elementary set of unary operations that readily support **Property B-1**, the consistent state property. To alleviate this difficulty, components should limit the impact of side-effects to a set sequence of parameter values. They may handle this either by aggregating all the operations into one that accepts an aggregate datatype, or by predicting the incoming values of a sequence and applying side-effects wisely. The former option eliminates the problems of side-effects, but places restrictions on the use of components and their operations; the latter option handles the problems of side-effects rationally, but burdens the components in handling their inputs with care.

In predicting the reception of a sequence, components should cache the inputs of unary operations, and when appropriate, apply the inputs collectively with side-effects. Only after a proper sequence is found should the side-effects occur. In determining that sequence, the components should not expect interactions to signal a beginning or an end; for interactions to do so would interfere with their role in coordinating communications. The handling of side-effects is a computational issue such that components should perform extra work in extracting the proper subsets from an indeterminate sequence of inputs.

To extract a proper sequence, components should recognize a complete sequence, search for repetitive inputs, handle domain specific predictors, and interact with strongly related operations. A complete sequence of inputs, such as \( \text{A, B, C,} \) is a clear indicator that components should update and produce side-effects. Further inputs unmistakably denote the beginning of a new sequence. Repetitive inputs end one sequence and begin another.
For example, the sequence "A, B, A'" separates into two sequences, one involving "A, B," and another involving simply "A'." Domain-specific predictors apply rules in separating inputs into individual sequences. In time-varying systems, for instance, a sequence stops whenever a component updates its temporal state. All changes produced by a sequence of inputs are instantaneous and can not span an interval of time.

In interacting with strongly related operations, components must resolve the difficulties that arise when side-effects affect the computations of other operations, such as those responding to external queries. For these operations, the components should terminate all sequences and induce side-effects. It is inappropriate for the components to ignore the open sequences and delay the introduction of side-effects. To do so would violate PROPERTY B—1, which requires components always to respond according to their current state. The unavoidable drawback to this response is that race conditions may inadvertently partition a single sequence into two. To avoid this drawback, it is best for components to issue warnings whenever terminating an open sequence. This problem will then be resolved by the proper coordination of scripts, not by the changing of components.

PROPERTY B—3: Components and their operations should support totality measures.

Totality requires components and their operations to produce well-defined results for all possible input values. By acting defensively, components ensure that they return meaningful values and that their operations perform correctly. Components must be responsible for reacting to all inputs because it is impractical to expect only valid inputs from the environment. Coordination activities are neither expected nor required to realize what range of inputs cause components to behave poorly. The coordination activities do not do so because information assessment is a computational, not an organizational, issue. Similarly, it is beyond the capabilities of components to determine whether the information they produce is acceptable for others since components neither acknowledge nor realize the identity of their communicating partners. Without such knowledge, it is impossible for components to evaluate the validity of the information they produce.

8.2.2 Contextual Example Revisited

It is also important that the components of the contextual example support the behavioral properties so that the components may be reused in multiple contexts. In observing PROPERTY B—1, which requires that components base their responses on past inputs, the robots and their controllers must compute their states carefully. This is especially critical for the
contextual example because the robots and their controllers ultimately effect the state of
the light. As illustrated in the diagrams of Figures 4.3 and 6.1, any information that inaccurately reflects the positions of the robots may toggle the light mistakenly. Having the robots respond appropriately is also helpful in debugging race conditions. If software interactions were required to maintain the consistency of the robots and their controllers, the interactions would need to monitor their execution carefully. If the interactions were too early or too late in updating the state of the components, the light would ultimately behave falsely.

In conformity with PROPERTY B—2 and its corollary, which requires that components provide a set of unary operations, the controllers partition their multi-parameter operations into many individual ones accepting single inputs. The controllers, for instance, provide separate operations for fixing their range, duration, method of interpolation, and current time. Any call to these operations updates a single parameter of the controllers and advances the state of the controllers to reflect their input continually.

In observing PROPERTY B—3, which states that components should support totality measures, the robots and their controllers act defensively when accepting input. The robots may receive hand positions that are beyond their reach, and the controllers may receive temporal values beyond their maximal duration. Although components may receive improper information, software interactions still require outgoing information to act properly. The information permits the remainder of the software network to operate properly, although not necessarily accurately. Future extensions to IBP may support exception handling, which would help in resolving problems that arise when invalid information is communicated between components.

8.3 Temporal

Temporal properties determine when components act and how components integrate their computations with time. They require that software components be time-invariant and that software components explicitly link their computations to $\Delta T$, a change in time. The first requirement permits interactions to synchronize easily the states of multiple components. Components that update their own perceptions of time burden software interactions with tracking their temporal state. The second requirement ensures that software components behave properly with respect to the advancement of time. In many software systems, the advancement of time is neither constant nor regular.

---

4In the context of this work, a time-invariant component may vary its state according to the passing of time. It may not, however, determine how time passes or apply any operators, such as those that scale, to its perceptions of time.
8.3.1 Properties

**Property T-1:** Components should be time-invariant.

A time-invariant component maintains state without advancing its own notions of time. Such a component operates within an environment in which time advances around it. The state of the component changes as the component receives notices from its environment. A time-invariant component is preferable to a time-varying one because an invariant one is relatively easy to coordinate. Coordination activities determine when and at what rate the component receives notices of temporal advancement. A component advancing its own notion of time is usually difficult to coordinate. It requires continual supervision to ensure that it operates neither too quickly nor too slowly in relation to other components. A time-invariant component is also preferable in developing asynchronous behaviors, such as those commonly found in parallel simulations [50]. Coordination activities may purposefully assign distinct timing values to cooperating components. Each component acts naturally, oblivious to the differing perceptions of its peers. The coordination activities work with the components to resolve differences that arise from temporal distortions.

**Property T-2:** Components should employ “explicit” means to link their computations to time.

In time-varying systems, a component may apply either explicit or implicit means to link its computations to the passage of time. With explicit means, a component applies time as an integral value of its computations. As time changes, the computations update state. With implicit means, a component infers time from the invocations of its operations. The sampling rate of the invocations, not advances in time, determines when and at what rate the computations update state. The differences between the two means are exemplified in the coding in Figure 8.4, which presents two methods for the component Foo. The operation `expBar` makes use of an explicit notion of time while the operation `impBar` operates with respect to an implicit notion of time.

There are two reasons to favor an explicit approach to time. First, an explicit approach supports the separation of computation and coordination. Coordination activities manage temporal progression while computational activities respond to changes in time. The two sequences of code in the latter half of Figure 8.4 illustrate how this separation simplifies the coding of time-varying systems. Each sequence updates the state of `z`, an instance of `Foo`. The sequence using `expBar` is far simpler than the sequence using `impBar`. The operation `impBar` requires its callers to control its evolution over time (line i2) and to pro-
Explicit

```cpp
void Foo::expBar( AT )
{
    x = x + AT * dx;
    y = y + AT * dx;
}
```

(e1) Foo z; int k=0;
(e2) z.expBar( 10 );
(e3) z.expBar( k );

Implicit

```cpp
void Foo::impBar()
{
    x = x + dx;
    y = y + dy;
}
```

(i1) Foo z; int k=0;
(i2) for(i=0; i<10; i++) z.impBar();
(i3) if (k != 0) z.impBar();

Figure 8.4: Explicit and implicit means for linking computations to time

tect the operation from invalid time steps (line i3). Second, an explicit approach produces components that readily support discrete-event simulation. Components react properly to advances in time that are neither periodic nor whole. For example, `expBar` in Figure 8.4 readily accepts fractional values for `k` (line i3) while `impBar` does not.

**Corollary T–2:** Components that employ implicit means should be bounded explicitly.

Occasionally, the application of an implicit approach to time by components is unavoidable. Some components, such as those that update incrementally, change state only when specific events arise. For these components, it is best to bind their updates to an explicit change in time. When time advances in great leaps, as could happen in some simulations, the components update their state accordingly. For example, if $\Delta S$ is the maximal time step bounding a state change, a component should update $X/\Delta S$ times when $X$ units of time pass. This integration of the two means for updating state provides two benefits. First, it permits components to place a lower bound on the number of times they ought to update over a period of time. The lower bound loosens the ties that relate state changes to implicit invocations and their sampling rate. The advancement of time contributes to the advancement of state. Second, it permits the components to provide an estimate of their future behavior. A sudden leap of time does not prevent components from computing an approximate state.

**Property T–3:** The operations of components that advance state with time or that facilitate synchronization should orchestrate their computations around $\Delta T$, a change in time.

There are two kinds of operations that accept temporal information, those that advance state and those that facilitate synchronization. Operations that advance state respond to notices of temporal advancement. These operations ensure that components remain con-
consistent over time. Operations that facilitate synchronization answer inquiries regarding temporal advancement. The inquiries ascertain whether the operations accept the next expected value of time. Time advances if all active operations provide a positive response to such inquiries.

The temporal information may arrive at the operations as either $\Delta T$, a relative change in time, or $T$, an absolute value of time. $\Delta T$ represents a time of temporal advancement or the elapsed time since an established starting time. The former $\Delta T$ indicates the amount of time that has passed since an operation was last accessed. This value of time works well for operations that perform numerical computations, such as differential equations. The latter $\Delta T$ indicates the amount of time that has passed since a specific activity has begun. This value of time works well for operations that base their calculations on a duration of time, such as functions that interpolate evenly over time.

Operations that accept temporal information may orchestrate their computations around either value of $\Delta T$. Each value of $\Delta T$ provides the same benefits. Both $\Delta T$’s permit operations to act non-linearly over time. They act at the rate at which they receive temporal information. For example, the two operations in Figure 8.5 act dissimilarly over time. Operation A acts more slowly because it receives fewer updates than Operation B. Neither operation acts as fast as global time does because neither receives continual updates. Both $\Delta T$’s also permit operations to disregard references to absolute time. Operations neither track nor compute temporal advancement. They permit coordination activities solely to determine when and how time advances.

\[
\begin{array}{c|c|c|c|c|c|c}
\text{Global} & \text{Advancement} & \text{Total} \\
\hline
a & \text{Operation A} & 12t \\
\hline
b & \text{Operation B} & 21t \\
\end{array}
\]

Figure 8.5: A timeline involving non-linear temporal advancement

Should an operation, such as those applying numerical integrators, expect to receive a specific timestep, they should adapt to accept any value of $\Delta T$. This enables the interactions of IBP to freely advance the progression of time and update simultaneously multiple operations with different expectations. The operations may adapt to an irregular timestep by applying an appropriate interpolatory function to estimate a new value. Upon reception of another timestep, the operation may update its state with regards to the time of its last
update or to the last time that its state was set precisely without the aid an interpolatory function. For instance, Figure 8.6 presents the state of an operation over a given set of \( \Delta T \)'s. At time 15, the operation estimates its current value. At the time 20, the time of its next update, the operation may apply one of two options to adjust its state. With the first option, the operation updates in accordance to its previous estimate (green line). The previous estimate is applied to its current computations. With the second option, the operation updates in accordance to the value of its state at time 10, the last time that the operation updated properly (red line). The second option ignores the previous estimate and applies only well-derived values of a previous time. The first option is far simpler, but less accurate. It is up to the discretion of the developer of the operation to decide which option is best.

![Figure 8.6: The use of interpolation to estimate the value of a fixed timestep integrator](image)

8.3.2 Contextual Example Revisited

It is important that the components of the contextual example observe the temporal properties so as to improve the components' reuse in contexts involving time. In observing Property T–1, which states that components should be time-invariant, the components, such as the controllers of the robots, maintain fixed states until accepting updates from temporal interactions. Normally, the interactions update the local times of the controllers so as to synchronize them to a global clock or to advance them forwards in time. In keeping with the global clock, the components support concurrent execution. In advancing ahead, the components provide future values, such as the future positions of a robot. Such positions are useful in enabling the robot to avoid collisions and in determining the robot’s momentum. The temporal interactions may also adjust the progression of time within the controllers. By varying the rate of temporal progression, the interactions may alter the speed of an entire simulation, or the just the speed of a single robot.

In following Property T–2 and its corollary, which provides that components should favor explicit means in updating state, the components adapt well to changes in
time. Time is integral to setting the states of the components or to establishing bounds on how often the components change state. For example, the controllers of the robots apply a parametric curve, such as that of Figure 8.7, in computing their states for a given value of time. The controllers extract a value from the curve by computing a parametric value, $\text{frac}$, with an explicit formulation or with a well-founded implicit form. The explicit formulation sets the value of $\text{frac}$ with a time-based formula while the implicit form bounds an update to $\text{frac}$ by $\Delta S$.

![Parametric Curve](image)

**A) Explicit formulation**

$$\text{frac} = \frac{\Delta T}{\text{duration}}$$

**B) Well-bounded implicit form**

$$\text{for (i=0; i<\Delta T; i += \Delta S)}$$

$\text{frac += .1}$

Figure 8.7: Two means for extracting a value from a parametric curve

The explicit formulation relates a change in time to a duration of time. The ratio between the two values computes the parametric location on the curve. As time progresses, the components interpolate the orientation and location of the robot smoothly from beginning to end. This explicit formulation is useful in moving robots whose movements are guided by the position of the light. Conversely, the implicit form relates a change in the parametric value to an implicit invocation. Each time the code executes, it updates the parametric value by a multiple of one-tenth of a unit. POSCONTROL, the controller moving ROBOTB, uses the implicit formulation to update the position of the robot that suffers from periodic breakdowns. For this robot, its working status, and not time alone, contributes to the functioning of its controller.

In complying with Property T-3, which states that components should synchronize their computations around $\Delta T$, the components operate non-linearly over time. For example, the two forms of Figure 8.7, as applied by the controllers in the contextual example, orchestrate their computations around $\Delta T$. The first integrates $\Delta T$ directly into its computations while the second uses $\Delta T$ to rationalize approximate changes in state. Each of the two forms readily accepts a variable rate of temporal advancement and provides an approximation of its future values.
This chapter presents the first of three ways to validate the applicability of IBS to visual simulation. It compares IBS directly with the tools of animation in terms of their effectiveness in supporting the use of several animating methods. The comparison provides empirical evidence of the utility of IBS in producing a wide range of scenes. The next two chapters present ways two and three. The second way demonstrates the utility of IBS in supporting the development of applications from different domains that rely upon animated visuals. The third way discusses the utility of IBS in improving the overall design of visual simulations.

9.1 Overview

The comparison of IBS with the tools of animation is achieved through three examples. Each of the examples consists of a general description and three separate implementations, produced from two representative tools and the RASP toolkit. The general description presents a simple scene exemplifying the use of a particular animating method. The scenes differ within the examples so as to present a range of diverse application of IBS to visual simulation, and more importantly, to present a problem that suits a particular method. The representative tools accompanying each example were chosen by their design and availability. They are representative in that they present a diverse spectrum of implementation approaches for visual simulation that apply a particular animating method. Such diversity enlists greater comparison between IBS and the tools of animation, and offers greater insights into the advantages and limitations of IBS. By comparing the size, adaptability, and flexibility of various implementations, the chapter shows how IBS provides greater support for applying
multiple animating methods, managing software complexity, and supporting software reuse.

9.1.1 Implementation and Analysis

The examples of this chapter use three animating methods common to visual simulation: key-frame animation, procedural animation, and particle system animation. Each method is useful to developers (and animators) in animating a wide variety of scenes. The three methods are representative of the current methods of animation in that each method uses a distinct set of data structures to apply a particular approach to animating state.

Of the three methods, key-frame animation is the most effective but the least efficient for producing an animation at a high level of precision. It applies controls for explicitly guiding the evolution of state changes over time. State changes occur as the system interpolates between the values of key states. Useful tools for evaluating the application of keyframing are MEL (ALIAS|WAVEFRONT’s scripting language) [6] and CORY (CLOCKWORK’s scripting tool) [56]. ALIAS is a widely popular, commercial software package, and CLOCKWORKS is an older yet representative system that proves useful in demonstrating the utility of keyframing and presenting its drawbacks.

Procedural animation is the most accurate but the least flexible of the three methods for fabricating an animation with precise results. Procedural animation involves the use of rules or computational algorithms in updating state values over time. It is most effective in developing animations that involve physically-based motions [77, 152] or recursive data structures [13, 29]. Useful tools for evaluating the application of procedural animation are MAM/VRS [36] and FRAN [41]. MAM/VRS is an object-oriented toolkit that separates the specification of geometry and behavior, and FRAN is a publicly available tool for interactive animation that, like this thesis, is based upon an unconventional method of programming and composes an animation from an assembly of parts.

Particle system animation is the most efficient but the least general of the three methods. It reproduces complex behaviors by abstractly describing the behaviors of many individual particles. The method updates the motions of a large number of particles to simulate the behavior of amorphous entities, such as water and fire, or of multi-entity crowds, such as flocks of birds or herds of animals. Useful tools for evaluating the application of particle systems are the PARTICLE SYSTEM API [96] and ASAS [127]. The PARTICLE SYSTEM API is a programming interface for the development of particle systems that separates the specification of particles and behavior, and ASAS is an experimental tool for multi-entity communications that separates the specification of behavior and time.
9.2 Key-Frame Animation

In the scene exemplifying the use of keyframe animation, three keyframes are applied to the movements of two objects, a ball and a camera. From simulation time 250 to 950, the camera and ball move independently along a "moving" path, initially established by an interpolation of the three keyframes. As the scene progresses, the entire path translates ten units along the x-axis in thirty units of time. The ball and camera travel along the path at different rates. Traveling twice as fast, the ball moves along the path in half the time. As the camera moves, it alters its orientation to observe the position halfway between the ball and 10 units above it. As shown in Figure 9.1, the camera's view continually changes as the positions of the ball and the path update over time.

![Figure 9.1: The camera adjusts its view as it follows the ball along a translating path.](image)

This example demonstrates the use of keyframing beyond that of strictly defining the values of key states. The example integrates the values of several keyframes and applies mathematical operators to them so as to produce a complex behavior. The integration and manipulation of keyframe values are techniques of increasing interest [168] since basic keyframing is extremely tedious to specify and arduous to apply when manipulating many values. Signal processing methods, such as sheering and filtering, are frequently applied with keyframing to produce new or alternative motions.

9.2.1 Cory’s Implementation

CORY [98], the scripting system of CLOCKWORKS, applies a two-tiered approach to keyframe animation. It partitions an animation into a hierarchical set of cues and scenes. Cues script the goals of individual actions while scenes script the performance of cues. An animation consists of several scenes, each acting independently and none overlapping in time.
In encouraging top-down design, only the scenes of animation access the global clock, which manages simulation time. The execution times of all cues are relative to the start of the scene that governs them.

The primary mechanism for producing key-frame animation is an evaluator, a time-varying value embedded into the definition of a cue. As time advances, the evaluator obtains a numeric value from a data-set, any function or object (such as that of a spline) that parameterizes its output to time. An evaluator modulates its value by modifying the output of its corresponding data-set. Modifications to the output are produced by altering a data-set's perceptions of time, such as shortening its duration, or by scaling the entire output of a data-set by a constant.

Building the Scene

The code of Figures 9.2, 9.4, and A.1 presents an implementation of the scene with CORY. The code of Figure 9.2 begins by creating a camera and a ball (lines 1-2) and establishing the properties of kScene, a scene consisting of three cues (lines 3-18). The scene starts at time 250 (line 7) and continues forwards in time for a duration of 700 units (line 8). As the scene progresses, it animates continuously the positions and orientations of the camera and ball with the three cues: cam_cue, trs_cue, and ball_cue.

   cam_cue updates the position and orientation of the camera with two actions, start.action and tick.action. start.action operates once to initialize four evaluators with timing information, which is local to the timing of the cue. The timing information decides when the four evaluators begin and for what duration they compute. tick.action executes repeatedly to provide the evaluators with timing values. The evaluators apply the timing values to compute their final states, which are derived from the outputs of their data-sets. The evaluators cam_x, cam_y, and cam_z apply the outputs of their data-sets twice. Initially, the three evaluators apply their data-sets in setting the position of the camera (line 30-32). Immediately afterwards, the three evaluators compute a midpoint between the position of the ball and the position 10 units above the camera (lines 33-41). The final lines of tick.action apply the midpoint with the formulas of Figure 9.3 in deriving the final orientation of the camera.

   trs_cue animates the location of the path, which guides the positions of the camera and ball, with two actions of its own. start.action initializes trns_x, an evaluator for computing a translational value (lines 48-50), while tick.action applies the same evaluator.

---
1For each evaluator, the attribute v.offset identifies an offset value while the attribute v.scale identifies a scaling value.
for updating the offset value of `cam_x`, the evaluator that determines the “x” coordinate value of the camera, and of `ball.x`, the evaluator that determines the “x” coordinate value of the ball (lines 51-53).

The code for cue `ball_cue`, which uses `ball.x` in animating the position of the ball, appears in Figure A.1 of the appendix. `ball_cue` is similar to `cam_cue` in that both cues
apply three evaluators to animate state. As set by kScene of Figure 9.2, the two cues differ in their starting times and durations (lines 7-10). ball.cue begins earlier and operates for a shorter period of time. Accordingly, the ball moves along the path twice as fast as the camera does.

A partial list of the code for establishing the evaluators appears in Figure 9.4. Each evaluator references a spline.data unit, a structure for keyframing that interpolates between key values. The spline.data units dataX, dataY, and dataZ are applied to animate the positions of the ball and camera while dataY2, replicating the data of dataY with a difference of ten, is applied to animate the orientation of the camera.

\[
\begin{align*}
\text{cam}\_x &= [(\text{cam}\_x \text{ value}) + (\text{cam}\_x \text{ v.offset})] \times (\text{cam}\_x \text{ scale}) \\
&= [(\text{cam}\_x \text{ value}) + (\text{ball}\_x \text{ value})] \times 0.5 \\
\text{look}\_y &= [(\text{look}\_y \text{ value}) + (\text{look}\_y \text{ v.offset})] \times (\text{look}\_y \text{ scale}) \\
&= [(\text{cam}\_y \text{ value} + 10) + (\text{ball}\_y \text{ value})] \times 0.5 \\
\text{cam}\_z &= [(\text{cam}\_z \text{ value}) + (\text{cam}\_z \text{ v.offset})] \times (\text{cam}\_z \text{ scale}) \\
&= [(\text{cam}\_z \text{ value}) + (\text{ball}\_z \text{ value})] \times 0.5
\end{align*}
\]

Figure 9.3: The formulas applied by Figure 9.2 in computing the orientation of the camera.

**Figure 9.4:** A partial implementation of the evaluators of Figure 9.2

---

2A complete listing of the evaluators appears in Figure A.2 of the appendix
9.2.2 MEL’s Implementation

MEL (Maya Embedded Language) is a scripting language that builds upon Alias|wavefront’s C++ API. The language permits developers to produce plug-ins and macros for MAYA, Alias|wavefront’s product for producing computer animation. MEL consists of imperative controls, such as loops and variable assignment, and of objects and function sets. Objects hold basic structures, such as geometry and keyframes, while function sets are C++ classes that operate on objects. The description of a scene normally contains several objects and various functions for animating that object over time.

MEL produces an animation from a scene’s description by collating the scene’s objects and function sets into a dependency graph, which MEL keeps hidden from the eyes of developers. State changes occur as the nodes of the graph enable communication between objects and function sets. To improve communications, MAYA evaluates the nodes of a graph lazily and progressively caches critical communications between nodes.

Building the Scene

The code in Figure 9.5 presents an implementation of the scene with MEL. The code begins by creating a camera and ball (lines 1-3) and keyframing their movements with an animation path consisting of three control vertices (line 4). The camera, ball, and path are represented as objects while the keyframed animation is set with setKeyFrame, one function of a function set for animating objects. Keyframing is also applied to the animation path so as to progressively translate the path along the x-axis (lines 5-6). The translation of the control vertices, as performed by the function pathAnimation, determines the shape and location of the path, and ultimately, the position and orientation of the camera and ball (lines 7-9).

To animate the viewpoint of the camera, a loop extracts the control vertices of the animation path and applies mathematical operators to them so as to compute the orientation of the camera (lines 9-20). The keyframing of the viewpoint, as set by setKeyframe, defines implicitly a path that traces the midpoints between the ball’s positions and the positions 10 units above the camera (lines 16-18). The commands to keyframe the location and

---

3 Alias|wavefront encourages its users to animate with MEL and MAYA’s graphical user interface. The current documentation for the C++ API is sparse and provides few examples. A comparison of the C++ API with that of IBS is worth investigating if and when the proper documentation avails.

4 In reality, MEL animates a viewpoint by controlling the orientation of a camera, not by explicitly keyframing the viewpoint. Hence, lines 12 to 15 of Figure 9.5 are fictional, producing error. Nonetheless, the code accurately conveys a working implementation of the scene, were MEL to provide such capabilities.
CHAPTER 9. IMPLEMENTING ANIMATING METHODS: A COMPARISON

create camera and ball
(1) camera -p 10 10 10 -name kamera;
(2) viewPlace -la 0 0 0 kamera;
(3) sphere -la 0 0 0 -name ball;

create and animate an animation path
(4) string $path = 'curve -d 2 -p 32 75 18 -p 32 75 -18 -p 16 22 50 -name cameraPath';
(5) setKeyframe -v 0 -t 0 -at tx $path;
(6) setKeyframe -v 10 -t 30 -at tx $path;

animate the position of the camera and ball
(7) pathAnimation -stu 250 -etu 950 -c cameraPath kamera;
(8) pathAnimation -stu 250 -etu 600 -c cameraPath ball;

initialize variables
(9) int $max = 'getAttr cameraPath.degree' + 'getAttr cameraPath.spans';
(10) int $time = 250;

produce animation curve for animating camera’s viewpoint
(11) for( $i = 0; $i < $max; $i++ ) {
(12) float $cv[] = 'pointPosition cameraPath.cv[$i]';
(13) $cv[0] = ($cv[0] + ... ) / 2.0;
(14) $cv[1] = ($cv[1] + 100) + ... ) / 2.0;
(15) $cv[2] = ($cv[2] + ... ) / 2.0;

animate eye point
(16) setKeyframe -v $cv[0] -t $time -at lkx kamera;
(17) setKeyframe -v $cv[1] -t $time -at lky kamera;
(18) setKeyframe -v $cv[2] -t $time -at lkz kamera;
(19) $time = $time + (700/$max);
(20) }

Figure 9.5: An implementation in MEL

orientation of the camera differ because the command pathAnimation applies only to an object’s position, not to its attributes. The viewpoint of a camera is an attribute and requires keyframing by explicit means, not by an association with an animation path.

9.2.3 RASP’s Implementation

The code in Figure 9.6 presents an implementation of the scene with RASP. The code consists of two objects, two interpolators, three virtual ports, and six software interactions. The two objects represent the ball and camera; the two interpolators keyframe the movements of the two objects; the three virtual ports affect the orientation of the camera; and the six interactions communicate and disseminate data and temporal information to the objects, interpolators, and virtual ports.

The definitions of the two objects and two interpolators appear at the top. The two objects, ball and kamera, are instances of Sphere and PerspCamera (lines 1-2). The two
create camera and ball
(1) GeoSphere *ball = new Sphere;
(2) Camera *kamera = new PerspCamera;

set keyframes with an interpolant
(3) EvolvePt *evK = new EvolvePt;
(4) evK->setMark( 0, Point3(32,75,18) );
(5) evK->setMark( 1, Point3(32,75,-18) );
(6) evK->setMark( 2, Point3(16,22,50) );

create interpolant to translate keyframe path
(7) EvolvePt *evT = new EvolvePt( Point3(0,0,0), Point3(10,0,0) );

create virtual ports for camera's viewing
(8) OPort *ev = *evK->outCurVal() + *evT->outCurVal();
(9) OPort *uplook = *ev + Point3(0,100,0);
(10) OPort *midlook = (*uplook + *ball->outPosition())/2.0;

establish temporal interactions
(11) DurationEvent *dEvt = new DurationEvent( evK->inDuration(), evT->inDuration() );
(12) TimeEvent *tEvt = new TimeEvent( evK->inCalcValue(), evT->inCalcValue() );
(13) TimeEvent *tEvt2 = *tEvt * 2.0;

establish software interactions
(14) Event *evt1 = new Event( ev, kamera->inPosition() );
(15) Event *evt2 = new Event( midlook, kamera->inLookAt() );
(16) Event *evt3 = new Event( ev, ball->inPosition() );

organize action unit
(17) Activity *act = new Activity( "moveCamera" );
(18) act->addInitEvent( dEvt );
(19) act->addActEvent( tEvt, evt1, evt2, tEvt2, evt3 );

organize time graph
(20) TmGroup *group = new TmGroup;
(21) TmInterval *inter = new TmInterval( 250, 950 );
(22) TimingAct *camMv = new TimingAct( act );
(23) group->addNode( inter, camMv );

Figure 9.6: An implementation in RASP

interpolators are evK (line 6) and evT (line 7). The first applies three keyframes to create a keyframe path (lines 4-6) while the second applies two positional values to translate the keyframe path along the x-axis.

As illustrated in Figure 9.7, the virtual ports uplook and midlook reference the ports of the two interpolators and the ball to determine where the camera peers as it moves along the translating path (lines 9-10). uplook computes a position 10 units above the position of the camera, and midlook computes a viewing position for the camera halfway between the position of the ball and uplook. The virtual port ev computes the position of the camera for uplook by summing the outgoing values of the two interpolators (line 8). The software interactions evt1 and evt3 also apply ev for communicating the positions of the ball and
camera over time.

Figure 9.7: The software network produced by the code of Figure 9.6

The script “moveCamera” orchestrates the execution of the six software interactions (line 17-19). It divides the six interactions into two groups, one consisting of durEvt (line 11), and another consisting of tEvt, evt1, evt2, tEvt2, and evt3. The first group executes once to set the durations of the interpolators while the second group executes repeatedly to advance the two interpolators in time and to animate the positions and orientations of the ball and camera (lines 14-16). Ordered sequentially in the second group, the interactions tEvt and tEvt2 update both interpolators with a single time step, such that tEvt2 doubles the time step of tEvt (lines 12-13). Each of the two interactions alters the local times of both interpolators so as to produce two different outgoing values for ev, which evt1 and evt3 use to animate the positions of the ball and camera. With separate time steps and the same interpolators and virtual ports, the ball and camera move along the same path at different rates.

The coding of the scene concludes with a small time graph that specifies the timing properties of act (lines 20-23). The script inherits the times 250 to 950 from inter, a node of type INTERVAL.

9.2.4 Comparison

An inspection of the three tools shows that each offers benefits and disadvantages to keyframing. All three share similarities in being a specification for animation, and all three differ in their approaches to software design, software development, and software reuse. The differences affect how the tools manage complexity and ease the development of animated scenes.

The implementations of CORY, MEL, and RASP are all similar in being specifications
for animation, not procedural definitions that produce immediate results upon execution. The execution of the three implementations produces intermediary forms that provide instruction to a run-time evaluator. The execution of RASP sequences software interactions; the execution of MEL organizes a dataflow graph; and the execution of CORY assembles cues into scenes. The run-time evaluator of each implementation examines the intermediary forms and executes them over time. The run-time evaluator, not the developers of animation, decides when and how the computer controls concurrent actions. Hence, the developers of animation are freed from explicitly embedding direct controls for parallelism, such as semaphores and co-routines, into the definitions of their keyframes. Moreover, the run-time evaluator is capable of integrating the values of several keyframes so as produce accumulative effects, such as that of moving a camera along a translating path.

Despite their similarities, all three tools apply a different approach to keyframing. With RASP, keyframing is applied by producing components and controlling their communications with interpolants. Small changes to either, which may be defined with interaction constraints or mathematical operators, produce new and interesting results. With MEL, keyframing is defined strictly with the application of function sets to objects, in which the communications between the two are defined implicitly. Thus, the communications between the two are difficult to vary so as to develop the communications most appropriate for a scene. For instance, the communications between pathAnimation and a camera are fixed to communicate only complete coordinate values. Hence, MEL is unable to keyframe the vertical coordinate value of a camera with only pathAnimation. Unlike what RASP permits with software interactions, MEL is unable to accept readily a single value and integrate it with the camera’s current position. Care must be taken either to extract the current position of the camera and integrate the position with new vertical values, or to control directly the dependency graph that interconnects the camera with pathAnimation. The former option provides pathAnimation with a complete set of coordinates while the latter requires the use of a additional function set that adds to the complexity of a script. With an additional function set, the script intermixes unsystematically two kinds of functions sets, one which abstractly manipulates the nodes of a dependency graph and another which manipulates directly the configuration of the same dependency graph.5

With CORY, keyframing is applied by partitioning a scene into many individual elements, each of which manages a separate concern. The approach is beneficial to CORY in

---

5The current version of MEL (3.0) does not document properly the use of function sets in manipulating a script’s dependency graph. A listing of all function sets indicates that such manipulations are possible, but documentation is lacking to show its use.
that it maintains a separation between the temporal specification of a scene and the application of computational elements. Similar to that offered by RASP, a hierarchy of abstractions decides how and when the computational elements perform over time. This approach differs from that of MEL, which binds tightly the temporal specification of a keyframed scene to the application of function sets with objects. The temporal specification is unavailable for independent update, analysis, and reuse. For instance, the temporal specification of “\texttt{pathAnimation -stu 250 -etu 950 -c cameraPath kamera},” which animates the camera from time 250 to 950 (line 7 of Figure 9.5), is inseparable from the command. To extract and apply the temporal specification elsewhere or to construct a temporal ordering similar to RASP’s time graph is impractical and nearly impossible.

The three tools also differ in their respective syntax. MEL applies a syntax that resembles an imperative scripting language, such as TCL [111]. Hence, it is probably the easiest of the three to learn. With MEL, developers freely decide the use and organization of objects and function sets. The drawback to this approach is that developers must also discover their own means for effectively managing complexity and supporting reuse. MEL provides few rules for introducing new objects and function sets, and few means for encouraging the reuse of its parts. For example, it is unclear how to manage the timing characteristics of the scene of Figure 9.2 at a high level of abstraction. It is also unclear whether the same scene can be encapsulated within an object so as to reuse that object and its keyframed values with the function sets of other scenes. The keyframed motions of the ball, for instance, could be useful for guiding the positioning of a secondary camera.

RASP and CORY each apply an unconventional syntax, and each involves a greater learning curve. To apply either tool successfully, one must understand IBS and the use of software interactions respectively, or comprehend the structure of cues and scenes. Building with CORY involves the use of declarative expressions. Developers assemble animations by parameterizing data structures with character strings and numerical values. The ordering of data structures is similar to the hierarchical ordering of RASP. However, the data structures of CORY are neither first-class nor organized via software composition. Hence, the structures neither support higher-order mechanisms for topology change, such as the meta-scripts of RASP, nor provide mechanisms for component design, such as the encapsulation of scripts into components. Hence, it is easier with RASP to coordinate the global effects of several keyframed scenes and to reuse the keyframed scenes in alternative contexts.

Despite its unconventional syntax, only RASP supports the development and use of keyframing with software composition. RASP encourages the development and reuse of a hierarchical organization of components. For instance, the organization of an action
CHAPTER 9. IMPLEMENTING ANIMATING METHODS: A COMPARISON

Graph builds hierarchies upon software interactions while a time graph builds hierarchies upon scripts. As noted by the Web3D community\(^6\) and the many developers of graphics tools [167, 145], hierarchies of components are easy to build and apply, and to share as a datafile. Moreover, the hierarchies can be applied separately to define, apply, and share keyframe values.

RASP is the only implementation of the three that supports rules for building components and reusing them effectively in a wide variety of contexts. Both RASP and Cory support a strong separation of concerns, but only RASP explicitly provides rules for maintaining that separation. To encourage reuse and minimize complexity, MEL simply suggests that all implementations not exceed fifty lines of code.\(^7\) With RASP, it is obvious how the components of the exemplary scene behave if time advances erratically and if multiple keyframing functions (or operations) attempt simultaneous updates to the components' states. The reuse guidelines of RASP associate the temporal states of all components to changes in time, not to absolute or implicit values of time, which may cause errors when time jumps erratically. Furthermore, the guidelines encourage the use of first-class operations, which components may apply to ensure that simultaneous updates caused by keyframing do not produce unwanted results.

9.3 Procedural Animation

In the scene exemplifying the use of procedural animation, a dexterous robot moves conditionally between two moving pucks, puck1 and puck2, for 200 units of time.\(^8\) The robot moves continuously between the two pucks until the animation ends. In between its movements, the robot waits at a puck for fifteen units of time. While the robot waits, it travels with the puck that lies underneath its feet. As shown in Figure 9.8, the robot's movements are linear as it moves from puck1 to puck2. When moving towards puck2, the robot leaps to a halfway point between the pucks. Before moving to puck2, the robot waits for the distance between the two pucks to be twelve units or less. If the distance is too great, the robot simply waits longer and travels further with puck1.

This example is exemplary in presenting procedural animation because it applies formulas and rules to the evolution of state. The formulas determine where the robot moves while the rules determine when the moves occur. The formulas are fitting because they are

\(^6\)To understand the views of the Web3D community, see the proceedings of the Web3D Symposium on Computer Graphics, and visit the URL “www.web3d.org.”

\(^7\)In practice, developers using MEL create scripts that far exceed fifty lines of code [48].

\(^8\)This scene is a subset of the scene described in Lee [88].
adequate but simple. To introduce greater realism, the formulas could easily be adapted to support physically-based motions. The conditions are fitting because they are simple, variable, and change with the state of the animation. The condition relating the distance of the pucks, for instance, changes with the movements of the pucks, and is valid only while the robot waits to move towards puck2.

9.3.1 MAM/Vrs’s Implementation

MAM/Vrs is an object-oriented toolkit, written in C++, that supports the specification of animation and 3D interaction with time-dependent functions and behavior graphs, logical hierarchies of nodes controlling shape and time. Implemented as objects, time-dependent functions animate by accessing the methods of graphical objects. As time advances, the functions supply the methods with state updating values. The behavior graph and its nodes direct an animation over time. Similar to those of the time graph of this thesis, the nodes of the behavior graph either calculate or control the advancement of simulation time, or build event-dependent constraints, which respond to specific events, such as the collision of two objects.

The nodes of the graph for controlling time are one of three types: temporal data types, time setters, or time modifiers. Temporal data types represent moments or durations; time setters specify or modify the timing requirements of computational elements; and time modifiers apply transformations to moments of time. The arrangement of the nodes outlines a plan for governing the performance of time-dependent functions.

Building the Scene

The code in Figures 9.9, 9.10, and A.3 implements the scene with MAM/Vrs. The code defines the scene as the building of XtRobot, a class with one constructor (Figure 9.9) and one
The constructor defines and places several time-dependent functions into a behavior graph, which the member function modifies in implementing the logic for animating the movements of the robot. The member function is periodically called upon by the behavior graph as a callback function.

```cpp
XtRobot::XtRobot()
{
    initialize scene
    SO<MSceneView> view = MSceneView(...);
    canvas = new TClCanvas(...);
    puck1 = new RCylinder(...);
    puck2 = new RCylinder(...);
    for moving robot (from puck1 to puck2) in a straight path
    mapLinear = new MTimeLinearMap<RVector>();
    moveToP2 = new MTimeCt<RComposite, const RVector&, RVector>(
        robot, &RComposite::setCenter, &RComposite::getCenter, mapLinear);
    moveToP2->setTimeRequirement(MTimeRequirement(0,0,30));
    for moving robot (from puck2 to puck1) along a curved path
    points = new RArray<RVector>(3);
    RArcCurve *curve = new RArcCurve(points->newIterator());
    MTimeCurveMap *mapCurve = new MTimeCurveMap(curve);
    moveToP1 = new MTimeCt<RComposite, const RVector&, RVector>(
        robot, &RComposite::setCenter, &RComposite::getCenter, mapCurve);
    moveToP1->setTimeRequirement(MTimeRequirement(0,0,30));
    for producing FSA behavior
    fsaBehave = new MBehaviorCallBack;
    fsaBehave->setTimeRequirement(MTimeRequirement(true));
    fsaBehave->setTimeCallback(new RMethodCallback(this, &XtRobot::fsaUpdate));
    adding behaviors to time graph
    studio()->addGeometry(canvas, view);
    studio()->addBehavior(moveToPuck2, moveToPuck1);
    studio()->addBehavior(fsaBehave);
    studio()->switchOff(moveToPuck1);
}
```

Figure 9.9: An implementation in MAM/VRS

The constructor for XtRobot initializes the scene (class) with several basic variables (lines 1-4) and two time-dependent functions, moveToP1 (lines 5-7) and moveToP2 (lines 8-12). The first function maps the movements of the robot to a straight line while the second maps the movements of the robot to a curved arc. Both the line and arc interpolate between several positions, which are unavailable to the constructor. The positions are only available when the scene executes. The constructor finalizes the scene by producing a third time-dependent function.

---

9 The code of Figure A.3 presents the class definition for XTRobot.
void XtRobot::fsaUpdate()
{
    
    ensures time event
    MTimeEvent *te = fsaBehave->currentTimeEvent();
    if (!te) return;

    computes time in current state
    double currentTime = te->getMoment().time();
    Bool bigTime = (currentTime - lastTime) > 15;

    controls operating state of robot
    switch( state ) {
        case 0:
            adjust linear path
            mapLinear->setBegin( robot->getCenter() );
            mapLinear->setEnd( puck2->getCenter() );
            if acceptable, advance to next state
            if (bigTime && abs( puck1->getCenter()-puck2->getCenter() ) <= 12) {
                lastTime = currentTime;
                studio()->switchOn( moveToP2 );
                studio()->switchOff( moveToP1 );
                state = 1;
            }
            break;
        case 1:
            adjust curved path
            RVector midpoint = ( puck1->getCenter() + puck2->center() ) / 2.0;
            points->setElement( 0, robot->getCenter() );
            points->setElement( 1, midpoint );
            points->setElement( 2, puck2->getCenter() );
            if acceptable, advance to previous state
            if (longEnough) {
                lastTime = currentTime;
                studio()->switchOn( moveToP1 );
                studio()->switchOff( moveToP2 );
                state = 0;
            }
            break;
        }
    }
}

Figure 9.10: The method called upon by the callback function of Figure 9.9

function (lines 13-15) and integrating all three functions into a behavior graph (lines 16-18). The third function, fsaBehave, uses the member function XtRobot::fsaUpdate to toggle the activity of the functions effecting the motions of the robot. Applying similar functions to that which initially disables moveToP1 (line 19), the member function switches the functions on and off to stagger their execution.

The member function fsaUpdate for XtRobot performs two tasks. It updates the positions of the motion paths (lines 26-27, 36-39) and toggles the movements of the robot
CHAPTER 9. IMPLEMENTING ANIMATING METHODS: A COMPARISON

The positions of the motion paths must change to accommodate the movements of the pucks. The positions of both pucks will undoubtedly change whenever the robot is in motion. The toggling of the robot’s movements occurs if simulation time is advancing (line 20-21) and the current state has been active for fifteen units of time (lines 22-23). If the robot is moving towards puck1, the toggling will not begin until the distance between the two pucks is less than twelve spatial units (line 28).

9.3.2 FRAN’s Implementation

FRAN is an experimental library for interactive animation that embodies a continuous model of time. Implemented as a Haskell library [68], FRAN embodies functional reactive animation, a declarative approach to building interactive content [41]. FRAN defines an animation as a collection of declarations, not as the by-product of a sequence of imperative commands. Consisting of data types and functions, the declarations compose time-varying values with events that trigger actions to animate state. For instance, the touching of a button by a user during an interactive animation could trigger the button to fade away.

The declarations of FRAN prevent developers from meddling with the details of piecing an animation together as a collection of parts and commands. For instance, a translating object is neither an object with an attribute that translates nor an object undergoing translation by external forces. The translating object is defined explicitly as an object that translates. The benefit of applying FRAN to animation is that its declarative definitions compose easily and encourage reuse.

Building the Scene

The code of Figures 9.11 and A.4 shows an approximate implementation of the scene with FRAN. The code is inexact because the current version of FRAN (v1.13) is insufficient to provide a complete implementation. The library lacks various functions, such as those for time management, and also lacks various operators, such as those for multiplying numbers and 3D points, which are critical for implementing the scene properly. Nevertheless, the code of Figure 9.11 begins with pseudo-code to define two moving pucks (lines 6-8). The remainder of the code proceeds to define a robot by its movements. The robot is defined declaratively as a geometric shape that follows a plan to alternate between two movements.

The code of Figure A.4 appears as the preamble to the code of Figure 9.11. The preamble defines two pucks of different sizes and colors, and one robot colored green. The two pucks effect the positional values of pos1 and pos2 (lines 6-7 of Figure 9.11).
The code concludes the animation by defining a display and associating the movements of the robot with those of the pucks (line 32).

```plaintext
animate position of the pucks
(6)  Point3B pos = origin3
(7)  Point3B pos1 = ... code to animate pos1 here ...
(8)  Point3B pos2 = ... code to animate pos2 here ...

plan for alternating the robot’s movements between pucks
(9)  actions :: UserB → Point3B → Point3B → GeometryB
(10)  actions u pos1 pos2 = (moveTo1 pos1) "untilB" stop1 =>> (moveTo2 pos2)
(11)  ‘untilB’ stop2 =>> actions u pos1 pos2
(12)  where
(13)  timeln = ... has been in active state for 15 units of time ...
(14)  dist1 = distance3 pos pos1
(15)  dist2 = distance3 pos pos2

cHECK CONDITIONS
(16)  cond = predicate (dist1 ==* 0) u
(17)  stop1 = cond &&* timeln

cHECK CONDITIONS
(18)  near = predicate (dist2 <* 12) u
(19)  stop2 = near &&* timeln

move robot towards puck#1
(20)  moveTo1 :: UserB → Point3B → GeometryB
(21)  moveTo1 u posl = (action u half 5) "untilB" cond =>> (action u posl 5)
(22)  where
(23)  half = (pos + posA) / 2
(24)  dist = distance3 pos half
(25)  cond = predicate( dist ==* 0) u

move robot towards puck#2
(26)  moveTo2 :: UserB → Point3B → GeometryB
(27)  moveTo2 u posB = (action u posB 10)

INTERPOLATE POSITIONAL VALUES
(28)  action :: UserB → Point3B → DoubleB → GeometryB
(29)  action u goal duration = moveTo3 pos robot
(30)  where
(31)  pos = (userTime u) / duration * goal

dISPLAY ROBOT AND PUCKS
(32)  main = displayU ((actions u pos1 pos2) "unionG" (puck1 "unionG" puck2))
```

Figure 9.11: An implementation in FRAN

The plan that guides the robot is produced with the function `actions`, which defines the conditions for advancing the robot towards a specific puck (lines 9-19). When the robot is situated at puck1 and has been advancing towards puck1 for at least fifteen units of time, the function moves the robot towards puck2 (lines 16-17). When the robot is within twelve units of puck2 and has moved towards puck2 for at least fifteen units of time, the function
repeats the robot’s original movement of advancing towards puck1 (lines 18-19).

The functions moveTo1 and moveTo2 define the advancing motions of the robot towards a particular puck (lines 20-27). moveTo1 simulates a leaping motion by advancing the robot to a halfway point, which raises the robot off the ground. Upon reaching the halfway point, the robot falls directly towards the puck (lines 20-25). Both functions produce their motions with the help of action, a function that interpolates the robot’s position (lines 28-31). It moves the robot from its current position to a goal position over a duration of time.

9.3.3 RASP’s Implementation

The code in Figure 9.13 presents an implementation of the robot’s movements with RASP. The code consists of one interpolator, seven software interactions, three virtual ports, and two virtual events. The interpolator computes the movements of the robot, which are determined by six interactions and one virtual port. The remaining virtual ports compute the distance between the pucks and the duration of the robot’s movements. These values and the virtual events are applied by an FSA, which is embodied by an interaction, to toggle the movements of the robot. As illustrated in Figure 9.12, the FSA contains two states, one for each of the robot’s movements. The code concludes by defining a script and embedding that script into a small time graph. The script updates fsa for a period of time, which is ultimately determined by the time graph.

The first five interactions communicate information to and from the interpolator (lines 2-6). Over time, the interpolator receives updates of time and puck positions from the interactions evt0 and jump. The virtual events stateEvt1 and stateEvt2 establish relations between evt0, the event providing the interpolator with time, and the events that parameterize the interpolator with puck positions (lines 9-10). The event fsa, a finite state automaton with two states, executes these virtual events to move the robot between pucks (lines 11-13). fsa uses stateEvt1 in its first state to initialize the interpolator with a linear
**interpolator**

1. EvolutePt *ev = new EvolutePt(3);

**parameterize the interpolator**

2. Event *eb = new Event( robot->outPosition(), ev->inBeginVal() );
3. Event *ef1 = new Event( puck1->outPosition(), ev->inFinishVal() );
4. Event *ef2 = new Event( puck2->outPosition(), ev->inFinishVal() );

**update the interpolator w/temporal values**

5. TimeEvent *evt0 = new TimeEvent( ev->inCalcValue() );
6. Event *eSet = new Event( ev->outCurVal(), robot->inPosition() );

**compute midpoint of robot’s jump to a puck**

7. OPort *mid = *(*(*robot->outPosition() + *puck1->outPosition())/2.0)+Point3(0,2,0);
8. Event *jump = new Event( mid, ev->inMidVal() );

**communicate midpoint of jump**

*establish virtual events*

9. VEvent *stateEvt1 = *ef2 && *evt0;
10. VEvent *stateEvt2 = *ef1 && *evt0 && *jump;

**establish declarative method w/finite state automaton**

11. FiniteStateEvent *fsa = new FiniteStateEvent( 1 );
12. fsa->setState( 1, eb, stateEvt1 );
13. fsa->setState( 2, eb, stateEvt2 );

**determine if robot is less than 12 units from puck2**

14. OPort *bClose2 = *distVP( robot->outPosition(), puck2->outPosition() ) < 12;

**does robot’s time in its current state exceed 15**

15. OPort *bTime = *fsa->outDurationCurState() >= 15;

**state transitions**

16. fsa->setTransition( 1, *bClose2 && *bTime, 2 );
17. fsa->setTransition( 2, 15, 1 );

**establish an action unit**

18. Activity *act = new Activity();
19. act->addActEvent( fsa, eSet );

**establish a time graph**

20. TmGroup *group = new TmGroup;
21. group->addNode( new TmInterval( 0, 200 ), new TimingAct(act) );

---

Figure 9.13: An implementation in RASP

...path (line 12). In its second state, fsa uses stateEvt2 to initialize the interpolator with a curved path (line 13).

As illustrated in the software network of Figure 9.14, jump uses the virtual port mid to provide the interpolator with an intermediate position for leaping (line 8). The intermediate position is produced by averaging the positions of the robot and puck1, and then applying a horizontal offset (line 7). The virtual ports bClose2 and bTime establish the conditions for advancing fsa to its second state, the moving of the robot towards puck2 (line 16). Produced with the function distVP, bClose2 returns true if the distance between the robot
and puck2 is less than twelve spatial units (line 14). bTime accesses a port from fsa and returns true if fsa spends more than fifteen units of time in its current state (line 15).

![Figure 9.14: The software network produced by the code of Figure 9.13](image)

### 9.3.4 Comparison

The implementations of MAM/VRS, FRAN, and RASP vary greatly in design. MAM/VRS and FRAN rely heavily upon user-defined functions and callbacks in creating a procedural animation. The functions either define the formulas that compute state or establish the conditions that control the evolution of state. RASP, in contrast, relies heavily upon the creation of scripts, which assemble a procedural animation as a collection of interactions and components. The interactions apply conditions to the transport of information, and the components apply formulas for computing the evolution of state. Reinforced with interaction constraints and multi-modeling methods, the definition of a procedural animation is confined to the body of a single script. Unlike the functions and callbacks of MAM/VRS and FRAN, the single scripts of RASP localize semantic information and develop from a methodic approach, two important traits in easing the development and application of procedural animation. The script for animating the movements of the robot procedurally, for instance, is easily changed to accept new formulas and conditions that determine when and how the robot moves.

Of the three tools, neither MAM/VRS nor FRAN provides effective mechanisms for controlling state changes with high-level abstractions or hierarchical structures. Apart from MAM/VRS providing a behavior graph, both tools encourage developers to create their own abstractions. Although the approach provides much freedom to developers, it is ineffective in promoting a consistent approach to procedural animation that is effective in managing complexity and encouraging software reuse. For example, the finite state automata (FSA)
that is embodied within the implementations of \textsc{Mam/Vrs} and \textsc{Fran} is difficult to apply in new contexts, as it is tedious to isolate the definition of the FSA from the code that toggles the movements of the robot. An FSA, which \textsc{Rasp} supports as an explicit abstraction, is a useful device for implementing other procedural animations, such as those that adapt the motions of a robot to its environment. On solid surfaces the robot walks, and on slippery surfaces the robot flies or swims.

\textsc{Rasp} and \textsc{Mam/Vrs} encourage a separation of concerns while \textsc{Fran} does not. The behavior graphs of \textsc{Mam/Vrs} are similar to the time graphs of \textsc{Rasp} in that both graphs manipulate the temporal aspects of a scene, independent of the nature of the scene's computations and the presentation of the scene's shapes.\footnote{\textsc{Mam/Vrs} provides a rich set of temporal operators that may prove useful for augmenting \textsc{Rasp}'s capabilities in managing time and computation separately.} \textsc{Fran} intentionally binds the description of components with the code that animates their behavior. With this coupling, the two concerns are difficult to extract or modify independently for use in other procedural animations. For instance, \textsc{Fran} embeds the timing of the example scene directly into the definition of the robot. The timing is neither easy to change nor easy to extract for use with other procedurally-defined movements, such as those having the robot swimming between two pucks. Furthermore, the timing, as it stands, is difficult to adapt because it accepts the use of temporal relations, expressions that schedule the timings of multiple actions. Procedural animation benefits greatly from the use of temporal relations in that the animation is capable of associating its actions with the execution of other actions.

### 9.4 Particle System Animation

In the scene exemplifying the use of particle system animation, one hundred particles slowly drift across a flat plane (see Figure 9.15). Moving for 500 units of time, the particles advance smoothly until they near a pole, which moves randomly on the surface of the plane. As the particles near the pole, they adjust their movements to avoid collision. When the particles are free from collision, they resume their original movements and drift freely over the plane. When the particles move beyond the boundaries of the plane, they terminate and begin again.

This example is representative of particle system animation because it moves many particles as a whole and applies commands to the particles so that they behave as a group. As a group, the particles move incrementally across the plane until they near the pole. Once they are near the pole, the group moves actively to avoid collisions.
9.4.1 ASAS's Implementation

ASAS is an object-oriented system that animates with the use of actors and newtons. Actors are independent computing processes that operate by way of message passing. They are initialized, activated, and terminated by scripts, themselves, or other actors. The actors of ASAS can respond to external state changes. Their behaviors are determined not only by their actions, but also by their relationship with their environment. Newtons are state variables that update automatically with the advancement of simulation time. Upon request, they interpolate between states according to the values of piecewise, continuous cubic curves. Newtons are especially useful for producing equations that repeatedly return new values.

Building the Scene

The code of Figures 9.16 and A.5 present an approximate implementation of the scene with ASAS.\footnote{The code of Figure A.5 appears as the preamble to the code of Figure 9.16. The preamble creates a moving pole and defines offset, a function for updating the position of a particle, such that the particle avoids the position of the pole.} The code is inexact because the documentation for ASAS is incomplete and the definitions of key functions are unavailable. To code the scene properly, the implementation of Figure 9.16 introduces set!, a new command for resetting the value of a variable. The semantics for set! is borrowed from SCHEME \cite{149}, a language for functional programming that shares many similarities with the original design and syntax of ASAS.

The code of Figure 9.16 begins with a script that animates 100 particles for 500 units of time (lines 10-16). The function create-particles recursively calls upon make-particle to model every particle as an actor (lines 17-21). The actor provides shape to a particle and animates its position over time. The parameters <rand?> initialize the local variable of make-particle, deltaV and position, with random starting values (lines 24-25). To control the position and movements of a particle, make-particle applies two conditionals. The first conditional compares the position of the particle with that of the pole. If the pole is nearby, the particle moves to a new position computed by the function offset, appearing in
**CHAPTER 9. IMPLEMENTING ANIMATING METHODS: A COMPARISON**

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(10)</td>
<td>(script moving-particles)</td>
</tr>
<tr>
<td>(11)</td>
<td>(local: (runtime 500))</td>
</tr>
<tr>
<td>(12)</td>
<td>(number 100));</td>
</tr>
<tr>
<td>(13)</td>
<td>(animate (cue (at 0)</td>
</tr>
<tr>
<td>(14)</td>
<td>(start (create-particles number)))</td>
</tr>
<tr>
<td>(15)</td>
<td>(cue (at runtime)</td>
</tr>
<tr>
<td>(16)</td>
<td>(cut))))</td>
</tr>
<tr>
<td>(17)</td>
<td>create many particles via recursion</td>
</tr>
<tr>
<td>(18)</td>
<td>(defop create-particles</td>
</tr>
<tr>
<td>(19)</td>
<td>(param: index)</td>
</tr>
<tr>
<td>(20)</td>
<td>(if (zerop index)</td>
</tr>
<tr>
<td>(21)</td>
<td>(then nothing)</td>
</tr>
<tr>
<td></td>
<td>(else (group-actors (make-particle) (create-particles (dif index 1))))))</td>
</tr>
<tr>
<td>(22)</td>
<td>produce one particle having shape, position, and velocity</td>
</tr>
<tr>
<td>(23)</td>
<td>(defop make-particle</td>
</tr>
<tr>
<td>(24)</td>
<td>(actor (local: (mote (recolor color particle))</td>
</tr>
<tr>
<td>(25)</td>
<td>(deltaV (vector &lt;randX&gt; &lt;randY&gt; &lt;randZ&gt;))</td>
</tr>
<tr>
<td>(26)</td>
<td>(position (vector &lt;randX&gt; &lt;randY&gt; &lt;randZ&gt;))</td>
</tr>
<tr>
<td>(27)</td>
<td>set particle</td>
</tr>
<tr>
<td>(28)</td>
<td>(grasp mote)</td>
</tr>
<tr>
<td>(29)</td>
<td>check if particle is near pole</td>
</tr>
<tr>
<td>(30)</td>
<td>(if (== (dist position pole)) poleRadius4)</td>
</tr>
<tr>
<td>(31)</td>
<td>(set! position (offset (dist position pole) position deltaV)</td>
</tr>
<tr>
<td>(32)</td>
<td>(else (set! position (plus position deltaV))))</td>
</tr>
<tr>
<td>(33)</td>
<td>check if particle moves beyond plane range</td>
</tr>
<tr>
<td>(34)</td>
<td>(if (&gt; position (vector -999 -999 215))</td>
</tr>
<tr>
<td>(35)</td>
<td>(set! position (vector &lt;randX&gt; &lt;randY&gt; &lt;randZ&gt;))</td>
</tr>
<tr>
<td>(36)</td>
<td>move and view particle</td>
</tr>
<tr>
<td>(37)</td>
<td>(move position)</td>
</tr>
<tr>
<td>(38)</td>
<td>(see mote))</td>
</tr>
</tbody>
</table>

Figure 9.16: An implementation in ASAS

Figure A.5. The second conditional determines whether the position of the particle exceeds the boundaries of the plane. If that occurs, the position of the particle is reset to a new initial value. The code for `make-particle` concludes with functions to update the position of the particle and to visualize the particle’s shape.

### 9.4.2 PARTICLE API’s Implementation

The PARTICLE API is a library of C++ functions for programming a particle system. Stylistically similar to the functions of OPENGL [105], which is a commonly accepted tool for graphical programming, the functions of the PARTICLE API base their behavior upon a global state. Every function call affects the global state and the performance of successive
operations. The functions of the PARTICLE API are of four kinds: state, group, action, and action list. Each kind produces a separate effect to the global state. State functions establish or modify the global state by controlling the attributes of particles. Group functions organize collections of particles into particle groups, which are then acted on by action functions. Group functions either add (or remove) particles from a group, or render the particles to a display. Action functions establish the behavior of a particle group. They produce effects, such as gravity and bouncing. Action list functions assemble action functions into scripts, such that the action functions apply easily to particle groups. The action list functions permit complex effects to be managed and applied like primitives.

Building the Scene

The code in Figure 9.17 presents an implementation of the scene with the PARTICLE API. The code begins with the creation of a particle group (lines 1-2) and with the preparation of an action list (lines 3-11). The particle group is identified with the handle particleJiandle. The action list contains five functions for establishing the attributes of the particles, such as their velocity and color (lines 5-7), and for determining where the particles roam in physical space (lines 8-9). Particles that roam beyond the physical space disappear and restart from the beginning. The code ends by setting the current time step and identifying a rendering loop, which determines how the particles avoid the moving pole and how the particles are drawn (lines 12-18). Each pass of the rendering loop instructs the particles of particleJiandle to avoid the current position of the moving pole and to appear on a display as line segments.

9.4.3 RASP's Implementation

The code appearing in Figures 9.18 and 9.19 presents an implementation of the scene with RASP. The code of Figure 9.18 animates a particle system with port groups, high-level abstractions that consolidate many individual ports into a single port providing or consuming multiple values. Established in the coding of Figure 9.19, the port groups represent the positions and velocities of every particle and of every motion controller. As illustrated in the software network of Figure 9.20, the code applies the port groups to establish three virtual ports to determine whether the particles are in close proximity to the pole (line 19-21). Particles are close if their distances are less than four times the radius of the pole (line 19).

The code uses four additional virtual ports to ensure that the movements of the particles avoid the position of the moving pole (lines 22-25). The virtual ports direct the
make a particle group
(1) int particle_handle = pGenParticleGroups(1, 100);
(2) pCurrentGroup(particle_handle);

create action list
(3) int action_handle = pGenActionLists(1);
(4) pNewActionList(action_handle);

set the attributes of particles
(5) pVelocityD( PDLine, 0,0,1.5, 0,0,2.5 );
(6) pColorD(1.0, PDLine, 0.8,0.9,1.0, 1.0,1.0,1.0 );
(7) pSize(1.0);

generate particles
(8) pSource(10, PDBox, -52,-130, 5,3.5,-105 );
   check if particles drift beyond plane
(9) pSink(false, PDPlane, 0,0,215, 0,0,-1);

move particles to their new positions
(10) pMove();

complete action list
(11) pEndActionList();

set time step, avoid pole, draw the particles
(12) pTimeStep( 1.0 );
(13) while(1) {
(14)    updatePolePosition();
(15)    pAvoid( 1, .1, 0, PDCylinder, 0,0,0, 0,pole.position,0, pole_radius,0 );
(16)    pCallActionList( action_handle );
(17)    pDrawGroup(GLLINES);
(18) }

Figure 9.17: An implementation with the PARTICLE_API

particles away from the pole in proportional increments by perturbing the current positions with random values. The code applies the virtual ports with two conditional software interactions, cEvt and cEvt2 (lines 27-35). cEvt applies one of two sets of software interactions for each particle. If a particle nears the pole, cEvt communicates the virtual port newPos to physicsP, the port group that references the particle’s position (lines 31-32). If the particle is not close, cEvt updates the position of the particle by communicating information from the motion controllers, motionP and motionV. cEvt2, performed after a particle moves, tests if a particle moves beyond the extent of the plane (lines 27-29). If so, the particle moves to the beginning and starts again. The code concludes with the definition of an action unit and a time graph. The action unit sequences the events tEvt and cEvt (lines 36-38). tEvt advances the motion controllers over time while cEvt controls the positioning of the particles. The time graph identifies an interval time for the entire animation and passes it onwards to the action unit (lines 39-42).

The code of Figure 9.19 is a partial implementation of the preamble to the code
check if particles are near pole
(19) OPort *minDist = *(pole->outRadius()) * 4.0;
(20) OPort *distance = dist( groupO->outAllPorts(), pole->outPosition() );
(21) OPort *isNear = *distance < *minDist;

compute new position of particles
(22) OPort *fact = *minDist / *distance;
(23) OPort *diff = *norm( *(groupO->outAllPorts()) - *(pole->outPosition() )) * *fact;
(24) OPort *off = *(diff + *(groupO->outAllPorts())) + *(rand->outVal());
(25) OPort *newPos = *off + *motionV->outAllPorts();

check if particles drift beyond plane
(26) OPort *offPlane = *(groupO->outAllPorts()) > Point3(-999,-999,215);
(27) CondEvent *cEvt2 = new CondEvent( offPlane );
(28) cEvt2->setTrueEvt( new CallEvent( rand2->inMakeVal()) );
(29) cEvt2->setTrueEvt( rand2->outVal(), groupI->inIndexPort() );

check if particles near pole
(30) CondEvent *cEvt = new CondEvent( isNear );
(31) cEvt->setTrueEvt( new CallEvent( rand->inMakeVal()) );
(32) cEvt->setTrueEvt( newPos, physicP->inOnePort() );
(33) cEvt->setFalseEvt( motionP->outIndexPort(), physicP->inOnePort() );
(34) cEvt->setFalseEvt( motionV->outIndexPort(), physicV->inOnePort() );
(35) cEvt->setFalseEvt( cEvt2 );

assemble activity
(36) TimeEvent *tEvt = new TimeEvent(motions->inTime());
(37) Activity *act = new Activity;
(38) act->addActEvent( tEvt, cEvt );

assemble time graph
(39) TimingAct *tAct = new TimingAct( act );
(40) TmInterval *interval = new TmInterval( 1, 500 );
(41) TmGroup *group = new TmGroup;
(42) group->addNode( interval, tAct );

Figure 9.18: An implementation in RASP

of Figure 9.18.13 The preamble organizes one hundred particles and one hundred motion controllers into several port groups, which assemble many ports into a single abstraction (lines 3-4). For instance, the port group groupO provides the positions of many particles by collating the ports of the particles that provide positional information (lines 3 and 12). All the positions are obtained instantaneously by accessing groupO's port outAllPorts; the function dist does this in ascertaining the distances between the particles and the pole (line 20 of Figure 9.18). Upon request, dist computes all the distances and stores them in the virtual port distance.

---

13A complete implementation of the preamble appears in Figure A.6 of the appendix.
CHAPTER 9. IMPLEMENTING ANIMATING METHODS: A COMPARISON

random number generators

(1) RandomPt3 *rand = new RandomPt3( Point3(-1,0,0), Point3(1,0,0) );
(2) RandomPt3 *rand2 = new RandomPt3( Point3(-5,2,-130), Point3(5,3.5,-105) );

assemble ports into groups

(3) OPortGroup *groupO = new OPortGroup;
(4) IPortGroup *groupI = new IPortGroup;

... load ports of particles into groups

(10) for(int i=0; i<100; i++) {
(11)  MotionLaw *motion = new LawsOfMotion;
(12)  Physics *pInfo = (Physics*) particles[i]->getAttribute( Physics::getClassTypeId() );
(13)  groupO->addPort( particles[i]->outPosition() );
(14)  groupI->addPort( particles[i]->inPosition() );
...}

Figure 9.19: A partial implementation of the preamble to the code of Figure 9.18

Figure 9.20: The software network produced by the code of Figure 9.18

9.4.4 Comparison

The implementations of ASAS, the PARTICLE API, and RASP differ in their approach to producing a particle system. ASAS treats every particle of a particle system as a separate entity with its own behavior, the PARTICLE API treats a group of particles as one entity with one behavior, and RASP treats many particles with individual behaviors as a single entity. In producing the scene, ASAS defines and animates many individual particles. Unlike RASP and the PARTICLE API, ASAS does not support a separation between the code that defines a
component and the code that animates its properties. Every particle contains the code that animates its behavior. Hence, to alter the behavior of the particle system, one must update the definition of a particle. Moreover, to reuse the behavior elsewhere, as in the guiding device for a flock of birds, one must extract its implementation from the particle.

In producing the scene, the PARTICLE API creates a particle group and selectively binds the group with attributes. This design manages complexity by accommodating hierarchical development and encourages reuse by separating the definitions of particles and attributes. Particle groups grow hierarchical as they embed other particle groups directly into their definitions. The PARTICLE API is useful but lacking in support for producing and applying particles systems of a wide variety. The tool offers no means for applying other animating methods, such as keyframing and procedural animation. To keyframe the movements of a particle system, developers must introduce their own measures, which may often be inconsistent across several applications. Furthermore, the API lacks a means for animating the attributes of particle groups or producing conditionals that decide when and how attributes apply. Particle systems with changing attributes produce fantastical results as they change their colors and behaviors over time. As developers do in using other animating methods, they must produce their own methods to vary the application of attributes. Most often, these methods involve the use of imperative commands that are troublesome to use in coordinating the behaviors of concurrent particle systems. For instance, the code in Figure 9.17 applies a “while” loop to update the particles with the positions of the moving pole. The while loop is well suited to handle particle systems that operate exactly in parallel but not those that begin and end at different times. When the times vary, many conditionals must be applied to ensure that every particle system operates properly.

The implementation of RASP applies an intermediary approach to the modeling of a particle system. It creates individual particles and collates them into a group with port groups. Although this design is less efficient than one that manages an entire group as a whole, like the PARTICLE API, the design permits the particles of a particle system to exhibit individual behaviors. Port groups are capable of collating many kinds of ports from many kinds of components. Each of the ports of a port group need only provide or consume the same type of values. For instance, the port groups of Figure 9.18 could easily accept a port from a component of type Fllocking, a behavior for modeling the movements of birds and fish. Every particle of this particle system would move similarly, except one that would act as though it was part of a flock. Furthermore, several port groups may be combined into one so as to collate the effects of several particle systems. Each of the particle systems act independently while being controlled and reused as a single group.
RASP is the only implementation of the three that readily permits adaptive time steps and enables concurrent particle systems to update at separate rates. Not every component or interaction of a particle system needs to update at the same rate or frequency. A particle group nearing collision may apply a smaller time step than a particle group moving freely in space. A smaller time step is often critical to avoid temporal aliasing, artifacts arising from the use of inappropriate time steps. ASAS and the PARTICLE API are less flexible in manipulating time in that they require the use of explicit controls in varying temporal progression. For instance, to animate two particle groups with separate time steps, the PARTICLE API requires the use of multiple "while" loops or the use of conditionals that compute separate time steps for each group.

9.5 Summary of Evaluation

This chapter has provided an empirical comparison between RASP and six different tools of animation: Cory, Mel, Mam/Vrs, Fran, ASAS, and the PARTICLE API. Each tool is representative of an alternative approach to animation. The comparison divided the tools into three sets and had each set implement an exemplary scene. The sets and scenes were chosen to present one of three animating methods common to visual simulation: keyframe-animation, procedural animation, and particle system animation.

The comparison demonstrates that IBS, as embodied by the RASP toolkit, is adept at applying multiple animating methods, supporting complexity measures, and facilitating reuse. IBS is capable of keyframing an animation and integrating its use with interaction constraints so as to apply controls to coordinate the evolution of state. The controls ease the use of operators in manipulating keyframe values, so that the keyframes offer greater reuse and are adaptable to run-time conditions. The software interactions of IBS freely apply keyframing to animate the states of all components, not only those components that enlist special functions.

IBS is also capable of animating a scene procedurally and integrating it with multi-modeling methods so as to guide the evolution of state with formulas and conditions. The multi-modeling methods localize the formulas and conditions that determine when and how an animation performs. The formulas and conditions are neither dispersed over multiple functions nor contained in callbacks. The methods also localize semantic information so as to facilitate the management of complexity. Moreover, the interactions and components that produce a procedural animation are readily accessible and easy to replace because they are confined to the definition of a script.
IBS is also capable of producing a particle system and integrating it with the reuse guidelines so that the behavior of particles varies over time. By communicating behavioral information to the particles, software interactions enable the particles to adapt to run-time conditions and join into groups that exhibit variable behaviors. Because the reuse guidelines allow a particle system to accommodate adaptive time steps, two particle systems may operate at separate rates.

The separation of computation from coordination, as supported by IBS and the general specification, is instrumental in facilitating the development of an animation and enabling its changes. Despite its absence in all but one of the six tools, this feature is quite useful in developing an animation from an assembly of parts and in controlling the timing of an animation independent of the behavior of its actions. As an assembly of parts, an animation is enabled to create hierarchies and to intermix sets of heterogeneous components. With separate controls, an animation is freely enabled to set the timing of its actions to vary with run-time conditions or the execution of other actions.
Chapter 10

Implementing Domain Applications: A Comparison

This chapter presents the second of three ways proposed by this thesis to validate the design and application of IBS to visual simulation. This chapter demonstrates the utility of IBS to support the development of applications from different domains that rely upon animated visuals for presenting results or for conveying time-varying behaviors. Two domains under investigation are algorithmic animation and scientific visualization. Each domain introduces a variety of concerns affecting the development and application of visual simulation.

In presenting its goals, this chapter applies a format similar to that of chapter 9. It presents an exemplary scene for a particular domain and compares the scene's development with RASP and with two other tools representative of the domain. The scenes demonstrate that IBS helps developers build applications in these domains. The designs of the application need not be altered radically to accommodate the use of IBS. In addition, each example provides further evidence of the benefits of IBS in supporting multiple animating methods, managing software complexity, and encouraging software reuse.

10.1 Overview

The examples of this chapter focus upon two domains in which visual simulation is key: algorithmic animation and scientific visualization. Each domain develops animation for a different purpose. Algorithmic animation uses animation to present a visual representation of the execution of an algorithm, such as one that sorts data or one that explores the structure of a complex data set. Those who build algorithmic animations encourage a strong separation between the code that animates and the code that defines an algorithm. The
CHAPTER 10. IMPLEMENTING DOMAIN APPLICATIONS: A COMPARISON

separation facilitates easy changes to each and simplifies the encoding of an algorithm, the execution of which is being animated [142]. Useful tools for comparing the application of algorithmic animation are POLKA [143] and OBLIQ-3D [104]. POLKA, the latest version of TANGO, is a popular tool for developing algorithmic animations, and OBLIQ-3D is a high-level, fast-turnaround 3D animation system that, like this thesis, bases its approach upon an unconventional method of programming.

Scientific visualization uses animation as a tool for observing the behaviors of dynamic systems and for presenting the values of time-dependent data sets, such as that of a weather pattern. As time progresses, an animation varies the position, representation, and attributes, such as color, of graphical entities representative of data sets and their states. Developers of scientific visualizations systems encourage the use of high-level abstractions for managing data sets and high-level functions for manipulating the values of the data sets. The data sets of scientific visualizations are usually large and are normally evaluated as a whole. Useful tools for building scientific visualizations are VTK [134] and VRML [159]. VTK is an open source, freely available visualization system that builds upon a dataflow architecture, the most commonly applied approach to scientific visualization. Each node of the dataflow architecture performs an elementary operation on a complete data set, for example, filtering the data set or extracting an iso-surface from its values. VRML is a general specification for interactive graphics that is gaining popularity in the visualization community [57]. It is viewable on the world wide web and is easily shared with others. In addition, VRML's approach to animation, which involves the movement of data between computational structures, is compatible with the dataflow architectures common in scientific visualizations.

10.2 Algorithmic Animation

In the scene involving algorithmic animation, twenty cylinders of varying heights animate "selection sort," a sorting algorithm that repeatedly swaps the smallest elements of an array to the front. As shown in Figure 10.1, each cylinder corresponds to a value and a position in the array. As the elements of the array swap to the front, their corresponding cylinders also exchange positions in 20 units of time. When two cylinders exchange positions, they follow a separate path so as to avoid collision. When the algorithm finishes, the array is sorted and the twenty cylinders appear in order, from smallest to largest.

The visualization of a sorting algorithm is the most widely applied example in presenting algorithmic animation. Sorting algorithms are commonly known, and their visualizations are clear and meaningful. The physical attributes of the cylinders, such as shape and lo-
CHAPTER 10. IMPLEMENTING DOMAIN APPLICATIONS: A COMPARISON

10.2.1 POLKA's Implementation

POLKA (Parallel program-focused Object-oriented Low Key Animation) is a 2½ dimensional animation system, written in C++, that visualizes the execution of algorithms and programs. A successor to TANGO [143], POLKA consists of a variety of procedures and abstractions for simplifying the animation of abstract concepts, such as variable swap and logical comparison. POLKA partitions an animation into three levels of abstraction: animator, view, and constituent. The animator level is the global manager of an animation. It executes visual effects in response to animation events, which are registered at the beginning of a program's body. The animation events identify the critical actions or phases of an algorithm, such as branching or swapping, that characterize the algorithm's behavior. The view level is the graphical perspective of a program. It controls the advancement of time and the presentation of visual elements. The constituent level is the low-level design of an animation. It animates the visual content of an animation with AnimObjects, such as lines and circles, and Actions, such as rotations and translations.

Building the Scene

The code of Figures 10.2, 10.3, and 10.4 presents an implementation of the scene with POLKA. The code consists of two class definitions and one main body. The two classes, My Animator and Rects, produce the animator and view level of the animation. The first class uses the

---

Figure 10.1: A visualization of selection sort

A 2½D image is a 2D image that employs visual cues, such as shadow drops and gradient shading, to simulate the appearance of 3D imagery.
second in interpreting the animation events of the main body. The second class produces the animation using AnimObjects and Actions, constituent level objects that produce rectangles and their movements.²

```
create user's Animator
(1) MyAnimator anim;

register events with Animator
(2) anim.RegisterAlgoEvt("Init",NULL);
(3) anim.RegisterAlgoEvt("Input","dd");
(4) anim.RegisterAlgoEvt("Display","d");
(5) anim.RegisterAlgoEvt("Exchange","dd");

initialize Animator
(6) anim.SendAlgoEvt("Init");

provide Animator with data
(7) for (int count=0; count<10; ++count) {
    (8) a[count] = int(rand()*10);
    (9) anim.SendAlgoEvt("Input", count, a[count]);
}

draw view of Animator
(10) anim.SendAlgoEvt("Show", 10);

perform selection sort
(12) for(int i=0; i<9; i++)
    (13) for(int j=i+1; j<10; j++)
        (14) if (a[i] > a[j]) {
            (15) swap(a[i], a[j]);
            (16) anim.SendAlgoEvt("Swap",i,j);
        }

Figure 10.2: An implementation in POLKA
```

The main body, shown in Figure 10.2, registers four animation events with `anim`, an instance of `MyAnimator` (lines 1-5). The animation events notify `anim` of key values, such as those of the array, and of key events that arise during the execution of the body. The code begins the sorting algorithm by initializing `anim` (line 6) and then creating an array of random values. The array values are provided to `anim` as animation events, referred to as "Inputs" (lines 7-10). The code concludes with an implementation of "selection sort," which sends an "Exchange" animation event to `anim` whenever two indices of the array exchange their values (lines 12-17).

The code of Figure 10.3 provides a partial implementation of the class interface for `myAnimator` (lines 18-34) and `Rects` (lines 35-52).³ `myAnimator` uses `Rects` to define and animate the movements of twenty rectangles. The class interfaces for `Rects` and `MyAnimator`

---

²Since the current version of POLKA does not support cylinders, the scene renders rectangles.
³A complete implementation of both classes appears in Figure A.7 of the appendix.
identify the correspondence between the animation events of the algorithm and the functions that animate the rectangles (lines 21-30). For instance, myAnimator interprets the animation event "Swap" by instructing Rects to swap the positions of two rectangles. The class Rects renders and animates the cylinders by storing an array of rectangles and maintaining a copy of the array that the main body sorts (lines 50-51).

```cpp
(18) class MyAnimator: public Animator {
(19) public:
(20) MyAnimator();
(35) class Rects: public View {
(36) public:
(37) Rects();

handle events
(21) int Controller() {
(22) if (!strcmp(AlgoEvtName,"Init"))
(24) r.Init();

(28) if (!strcmp(AlgoEvtName,"Swap"))
(29) r.Swap(AnimInts[0], AnimInts[1]);

(31) }
(32) private:
(33) Rects r;
(34) }
(35) 
```

Figure 10.3: An implementation in POLKA

The code of Figure 10.4 defines Ready and Swap, the two functions of Rects that are responsible for animating the movements of the rectangles. Ready creates the rectangles with RectangleGroup, an abstraction for producing a column of rectangles. The initial values for the RectangleGroup determine the column's attributes, such as size and spacing (lines 53-63). Ready controls the timing of the animation by indicating that each rectangle animates for a duration of 20 units of time (lines 64-66). Swap animates the exchange of two rectangles by accessing their current positions, computing a midpoint, and forming three actions (lines 68-72). The three actions, defined as m1, m2, and mid, apply the two positions and the midpoint in moving the rectangles in opposite directions. One rectangle moves directly from start to finish, while the other moves from start to finish via the midpoint (lines 73-79). Swap concludes by swapping the values of the array that reference the rectangles. Rects must swap the variables to maintain a correspondence between the array that it applies in producing rectangles and the array that the main body sorts (lines 80-82).
int Rects::Ready(int n)
{
    create rectangles
    RectangleGroup objs;
    initialize rectangles
    strcpy(objs.color,"black");
    objs.spacetype = SpacingNone;
    objs.horiz = 1;
    objs.align = AlignBottom;
    objs.useints = 1;
    objs.intvals = values;
    objs.intmin = 0;
    objs.intmax = max;
    objs.Make(this,
               blocks,n,.l,.l,.9,.9);
    rectangles appear now
    for (int i=0; i<n; ++i)
        blocks[i]->Originate(0);
    draw rectangles
    time = Animate(time,20);
    return(20);
}

int Rects::Swap(int i, int j)
{
    get locations & compute midpoint
    LocPtr loc1 = blocks[i]->Where(PART_SW);
    LocPtr loc2 = blocks[j]->Where(PART_SW);
    int x = loc1->XCoord()+loc2->XCoord())/2.0;
    int y = loc1->YCoord()+loc2->YCoord())/2.0;
    LocPtr mid = new Loc( x, y + .8);
    produce movements
    Action ml("MOVE",loc1,mid,7);
    Action m2("MOVE",mid,loc2,7);
    Action mid("MOVE",loc2,loc1,7);
    apply movements to rectangles
    int len = blocks[i]->Program( time,&ml );
    len += blocks[i]->Program( time+len,&m2 );
    blocks[j]->Program( time,&mid );
    time = Animate( time,len );
    swap values of internal array
    Rectangle *tempr = blocks[i];
    blocks[i] = blocks[j];
    blocks[j] = tempr;
    return(len);
}

Figure 10.4: An implementation in POLKA

10.2.2 OBLIQ-3D's Implementation

Derived from Modula-3 [26], OBLIQ-3D is an animation system that couples Obliq, an object-oriented, lexically scoped, untyped, interpreted language with Anim3D, an animation library with a 3D programming interface. OBLIQ-3D animates a scene by creating views, abstractions that encapsulate the definition of graphical objects with time-variant properties. Graphical objects consist of geometric shapes, lights, and cameras. Time-variant properties are functions of time that describe the attributes of the graphical objects. As time advances, the attributes of the graphical objects vary. The time-variant properties that animate state are of three kinds: asynchronous, synchronous, and dependent. Asynchronous properties vary independently, synchronous properties vary when signaled, and dependent properties vary continuously in accordance to the changes of others.
Building the Scene

The code segments of Figures 10.5 and 10.6 present an implementation of the scene with OBLIQ-3D. The first segment implements "selection sort" while the second defines the animation’s view. In implementing the algorithm, the sorting procedure notifies the view when the columns of the animation should exchange positions.

```
(1) let alg = proc( view )
(2) let C = [ {t=>2.5,id=>0},{t=>1.0,id=>1},{t=>2.0,id=>2},... ];
(3) view.start( #(C) );
(4) for i=0 to #(C)-1 do
(5)   view.addColumn( i, C[i].t );
(6) end;

    perform selection sort
(7) var ii = 0, jj = 0;
(8) for i=0 to #(C)-2 do
(9)   for j=i+1 to #(C)-1 do
(10)  ii := getIndex( C, i );
(11)  jj := getIndex( C, j );

        swap columns and indices
(12)  if C[ii].t > C[jj].t then
(13)    view.swapColumns( jj, ii );
(14)    swapIndices( C, ii, jj );
(15)  end;
(16) end;
(17) end;
(18) end;

    add view to display
(19) RootGO_NewStd().add( view.scene );
(20) alg( view );
```

Figure 10.5: An implementation in OBLIQ-3D

The code in Figure 10.5 implements "selection sort" by initializing an array of values and parameterizing the animation’s view with columns of a particular index and height (lines 2-6). Each column retains an index that corresponds to its location within the array. The sorting algorithm calls upon `getIndex` and `swapIndices` to swap the indices of columns that have their positions switched in the array (lines 10-11,14). The method `getIndex` locates the column whose position corresponds to a specific index of the array. The algorithm instructs the view to update the positions of two columns by calling upon the method `swapColumns` (line 13). The concluding statements of Figure 10.5 add the view to a root object and instruct the sorting algorithm to begin (line 19-20). The root object is a special kind of

---

4The coding of `getIndex` and `swapIndices` appear in Figure A.8 of the appendix.
CHAPTER 10. IMPLEMENTING DOMAIN APPLICATIONS: A COMPARISON

graphical object that creates a visual display.

The code of Figure 10.6 implements the view of the animation as view, an object storing three variables and three functions. The three variables define a scene, an animation handle, and an array of columns (lines 22-24). The scene assembles the columns into a single group, the animation handle controls the movements of the columns, and the array stores references to individual columns. The three functions of view are start, addColumn and swapColumn. start initializes the view (lines 25-28) while addColumn increasingly augments the view with a new column created from two time-variant properties, bottom and top (lines 29-36). bottom synchronizes its value with the animation handler while top forms a dependency with bottom. The two properties control the orientation of the column so that the column always remains vertical, regardless of its location. When bottom changes its value in response to a call from the animation handler, top adjusts its value to place itself directly above bottom. swapColumn animates the movements of two columns, which are selected in the function’s argument list (lines 37-45). The function swaps the locations of the columns by having each column move towards the position of the other. The first column travels a path close to the midpoint while the second moves in a straight line.

10.2.3 RASP’s Implementation

The code of Figures 10.7 and 10.8 presents an implementation of the scene with RASP. The code in Figure 10.7 implements the sorting algorithm with conventional programming constructs while the code in Figure 10.8 scripts the movements of two cylinders with keyframing. The execution of the algorithm progressively produces a time graph that contains multiple instances of swapAct, the script for animating the movements of the two cylinders. By its position in the time graph, each instance of swapAct inherits an interval of time that begins when the previous interval ends. Thus, the swapping of columns occurs in succession, which emulates the behavior of the sorting algorithm.

The code of Figure 10.7 animates “selection sort” by associating geometric columns with the indices of an array and by progressively producing a time graph that presents the algorithm’s execution. Each successful comparison applied by the algorithm adds two nodes to the time graph (lines 12-13). The first node, of type TmShift, updates an inherited interval by 20 units of time. The second node, of type TimingAct, maintains a reference to swapAct, the script of Figure 10.8 that animates the exchange of two cylinders (line 11). Collectively, the two nodes specify that swapAct begins with the end of the previous script and that swapAct animates for 20 units of time. The procedure getModelWithIndex returns a column with a particular index within the array (lines 4-5). The index associated with a
establishes view

let view = {
  scene => GroupGO_New(),
  ah => AnimHandle_New(),
  columns => ok,
}

initialize the view

start => meth( self, m )
  self.columns := array_new( m, ok );
  self.scene.flush();
end,

adds columns to view

addColumn => meth( self, u, height )
  let bottom = PointProp_NewSync( self.ah, [u-2,0,0] );
  let top = PointProp_NewDep( meth(self,time),
    Point3_Plus( bottom.value(time), [0,height,0] ),
    end );
  self.columns[u] := CylinderGO_New( bottom, top, 0.2 );
  self.scene.add( self.columns[u] );
end,

swap the positions of two columns

swapColumns => meth( self, u, v )
  let pv = self.columns[u].getProp( CylinderGO.Point1 );
  let pu = self.columns[v].getProp( CylinderGO.Point1 );

  move first column via midpoint
  let p = pu.get();
  let m = ( pu.get() + pv.get() ) / 2.0;
  pv.getBeh().linMoveTo( [m[0],m[1]+5,m[2]], 0, 1 );
  pv.getBeh().linMoveTo( [p[0],p[1],p[2]], 1, 1 );

move second column directly
  let q = pv.get();
  pu.getBeh().linMoveTo( [q[0],q[1],q[2]], 0, 2 );
  self.ah.animate();
end);

Figure 10.6: An implementation in OBLIQ-3D

column corresponds to a position in the array being sorted, not to the position within the array that stores the columns.

The code of Figure 10.8 animates the movements of two cylinders by using two interpolators, eight software interactions, and two virtual ports. The two interpolators swap the positions of the cylinders to visualize the exchange of information (line 21-22). As illustrated in the software network of Figure 10.9, the interactions iEvt1 and iEvt2 operate inversely when parameterizing the two interpolators with a beginning and an end (lines 23-24). iEvt1 communicates the positions of the two cylinders as values to begin and end the
(1) Model *bld1, *bld2;
(2) TmGroup *group = new TmGroup;
(3) group->addNode( new TmInterval(20) );
    performs “selection” sort
(4) for(i=0; i<MAX-1; i++) {
(5)     for(j=i+1; j<MAX; j++) {
(6)         bld1 = getModelWithIndex(i);
(7)         bld2 = getModelWithIndex(j);
(8)         float h1 = getHeight( bld1 );
(9)         float h2 = getHeight( bld2 );
    compare values and swap if needed
(10)        if (hi > h2) {
(11)            sort = swapAct( bld1, bld2 );
(12)            group->addNode( new TmShift(20) );
(13)            group->addNode( new TimingAct(sort) );
(14)            swapIndices( bld1, bld2 );
(15)            k++;
(16)        }
(17)    }
(18) }

Figure 10.7: An implementation in RASP

first interpolator; iEvt2 communicates the same two positions as values to end and begin the second interpolator. The virtual ports ctr and mid compute an intermediary position for the first interpolator to use in computing a collision-free path for the first cylinder (lines 25-26). The interaction mEvt communicates the intermediary position to the interpolator when the animation begins (line 27).

The concluding statements divide the eight interactions into two groups, initial and acting (lines 32-34). The initial events parameterize the interpolators with durations and initial values. durEvt, for instance, parameterizes both interpolators with a single interval of time (line 28). The acting events update the interpolators with time and apply the interpolators to animate the positions of the cylinders. timeEvt, for example, updates continuously the times of both interpolators (line 29).

10.2.4 Comparison

The implementations of POLKA, OBLIQ-3D, and RASP are all similar in that each maintains a separation between the code that animates and the code that implements an algorithm. Hence, each permits developers to apply modest changes to each part without requiring significant changes to the other. For instance, the code applied by each tool for animating
CHAPTER 10. IMPLEMENTING DOMAIN APPLICATIONS: A COMPARISON

Figure 10.8: An implementation in RASP

```c
Activity* swapAct( Model *bld1, Model *bld2 )
{
    GeoCylinder *cyl = (GeoCylinder*) bld1->getInfo( GeoCylinder::getClassTypeId() );
    GeoCylinder *cyl2 = (GeoCylinder*) bld2->getInfo( GeoCylinder::getClassTypeId() );
    EvInOutPt *evol = new EvInOutPt(3);
    EvInOutPt *evol2 = new EvInOutPt(3);
    Event *iEvt1 = new Event( bld1->outPos(), evol->inBegin(), evol2->inFinish() );
    Event *iEvt2 = new Event( bld2->outPos(), evol->inFinish(), evol2->inBegin() );
    OPort *ctr = *(bld1->outPosition() + bld2->outPosition()) / 2.0;
    OPort *mid = *ctr + Point3(0,40,0);
    Event *mEvt = new Event( mid, evol->inSetMark() );
    DurationEvent *durEvt =
        new DurationEvent( evol->inDuration(), evol2->inDuration() );
    TimeEvent *timeEvt = new TimeEvent( evol->inCalcValue(), evol2->inCalcValue() );
    Event *eEvt1 = new Event( evol->outCurVal(), bld1->inPosition() );
    Event *eEvt2 = new Event( evol2->outCurVal(), bld2->inPosition() );
    Activity *act = new Activity;
    act->addInitEvent( durEvt, iEvt1, iEvt2, mEvt );
    act->addActEvent( timeEvt, eEvt1, eEvt2 );
    return act;
}
```

the movements of two columns works equally well in animating other sorting algorithms, such as bubble sort. Each implementation differs, however, its approach to structuring an animation. RASP applies a hierarchical approach, while POLKA and OBLIQ-3D do not. The hierarchical structures of RASP, embodied by scripts and time graphs, present a logical and concise, though general, structure to an algorithmic animation. To animate the movements of the columns, POLKA and OBLIQ-3D define multiple functions while RASP creates only one. With POLKA and OBLIQ-3D, the timing of the animation is distributed over several functions, none of which handle time alone. Thus, it is not readily apparent how the timing of the animation may be changed. Changes to the timing of an algorithmic animation are important in analyzing the algorithm’s behavior and in presenting its features visually.

Each implementation also varies in its approach to animating state. RASP applies its structures, ports and interactions, uniformly across all components, while POLKA and OBLIQ-3D apply their structures, animators and views, uniquely to individual components.
In RASP, software interactions apply equally in communicating information between the ports of all components. In POLKA, components, such as those representing a column, are defined to contain the code that animates their state. The components update their states in reaction to events signaled by their environment. For example, Rects of Figure 10.4, the code that represents the columns, contains code to animate the exchange of columns. The coding of the columns and the coding to animate the columns are difficult to separate for reuse in other algorithmic animations. In OBLIQ-3D, components only animate if they are declared specifically to do so. To support animation, the components must bind their attributes to time-variant properties that exhibit only one kind of behavior at any given moment. To alter the way that components animate over time, the attributes of the components must bind with new properties, or the existing properties must be adapted to change their behaviors. This limitation hinders OBLIQ-3D from easily animating similar algorithms of greater complexity, such as quicksort, which animates the movements of cylinders whenever it swaps the values of its array or whenever it splits its array in half.

RASP is the only implementation of the three to provide precise controls for managing time and for associating time with animating actions. Such controls are important in varying the timing of an algorithmic animation, especially one that involves concurrency or sequential processing. For instance, with POLKA, it is difficult to delay the timing between successive actions. A delay, especially a variable one, is useful in adapting an animation to visualize additional behaviors, such as comparing two values before the exchange of columns. With RASP, delays are produced easily with changes to the time graph. OBLIQ-3D controls time and the scripting of animating actions with an animation handler, of which there may be several defined in one animation. At any moment in time, only one animation handler
may be active. Thus, it is difficult to specify two independent actions as being concurrent unless they are produced with the same animation handler. For parallel algorithms, such as quicksort, this restriction increases the complexity of their visualizations. Parallel actions may not be produced independently without referencing a common animation handler.

RASP and OBLIQ-3D share similarities in that each applies constraints to reduce the complexity of coding an algorithmic animation. However, their placement and use of constraints differ. RASP applies time-varying constraints to the ports of components while OBLIQ-3D applies fixed constraints to the properties of attributes. Each implementation offers its own benefit. Time-varying constraints may change over time. Hence, a port may experience several constraints and animate differently as an algorithm; for example, one applying randomization progresses through several phases. Fixed constraints, unlike time-varying constraints, are beneficial in that they update automatically. The constraints of RASP, by contrast, update only after a software interaction requests a value. With fixed constraints, an algorithmic animation applies less code in maintaining a correspondence between the parts that animate and the parts that implement the algorithm.

10.3 Scientific Visualization

In the scene involving scientific visualization, a sphere representing a complex data set and its bounding box are animated to change visibility, position, and color. The bounding box appears in place of the sphere when the representation of the sphere is unnecessary. This example is representative of the visualization of physical elements, such as in medical imaging, when it is often useful to switch repeatedly between two representations of a complex shape. Although a complete representation is critical for presenting shapes in close proximity to a camera or nearby surfaces, a bounding box is an acceptable replacement when the exact form of a shape is of secondary interest or is simply unwanted. The rendering of complex shapes can introduce rendering lags and may also clutter an image with extraneous information. For example, in animating the human heart, an exact rendering of the human lungs obscures the visibility of the heart and its movements.

As illustrated in Figure 10.10, the scene begins with the sphere appearing alone from time 0 to 90. At time 90, the bounding box appears gradually while fading from black to red. At time 100, the sphere disappears and the transition of the bounding box completes. Immediately following, the bounding box moves vertically for twenty spatial units. When

---

5Future work applying fixed constraints to RASP may prove beneficial to enhancing IBS and further simplifying the management of complexity.
the vertical position of the bounding box falls within the range of five to ten units, the sphere re-appears instantaneously as a temporary replacement for the bounding box. As the sphere travels outside the range, the bounding box re-appears and the sphere vanishes. At time 200, the bounding box halts and the visualization ends.

![Figure 10.10: A scene exemplifying scientific visualization](image)

Although this scene is simple, it demonstrates the difficulties of using a dataflow architecture for animation. Simple changes, such as those involving an object’s position, representation, or color are useful in presenting the dynamics of a system. Dataflow architectures, such as those applied by the visualization community, provide few means to alter the connections between nodes that are critical to animating state.

10.3.1 VTK’s Implementation

VTK (Visualization ToolKit) is a freely available, object-oriented, multi-language tool for scientific visualization. Like Pv-Wave [158] and other commercial systems, VTK applies a dataflow architecture to visualize scientific data. Modules, performing algorithmic operations, are bound to each other with connections so as to process and to visualize graphical elements. The connections among the modules produce a network that operates under implicit controls. Every module of the network requests data from its connections to ensure that its outputs accurately reflects its inputs. To animate a network and its modules, VTK relies upon the facilities of a host language, such as C++, to provide constructs for looping and conditional branching.
Building the Scene

The code of Figures 10.11 and 10.13 implement the scene with VTK. The first figure constructs the scene with high-level abstractions that form the nodes of a dataflow architecture. Data flows independently between the inputs and outputs of components via function calls. For instance, a sphere is produced by linking an abstraction that produces a data set with another abstraction that builds a polygonal representation. The second figure uses the statements of C++ to manage the connections between components. The statements introduce temporal controls and conditionals to manage the connections over time.

The code of Figure 10.11 constructs the scene with seven abstractions that interconnect to form a renderer, a sphere, and a bounding box. The renderer visualizes the two shapes within a graphical window (lines 1-3). As illustrated in Figure 10.12, the implementation of both the sphere and the bounding box derive from the management of dataflow between sphereSource and a renderer. sphereSource produces a data set for sphMapper, which produces a polygonal data set for sphActor, the structure representing the sphere (lines 4-9). The bounding box is produced similarly, but with one additional abstraction. It applies outline, a filter that bounds data to a box, in providing data for outlineMapper, which interconnects with outlineActor, the structure representing the bounding box (lines 10-16).

```
create the vtk renderer stuff
(1) vtkRenderer *ren = vtkRenderer::New();
(2) vtkRenderWindow *renWin = vtkRenderWindow::New();
(3) renWin->AddRenderer(ren);
create a vtkSphereSource object
(4) vtkSphereSource *sphereSource = vtkSphereSource::New();
(5) vtkPolyDataMapper *sphMapper = vtkPolyDataMapper::New();
(6) sphMapper->SetInput(sphereSource->GetOutput());
(7) vtkActor *sphActor = vtkActor::New();
(8) sphActor->SetMapper(sphMapper);
(9) sphActor->GetProperty()->SetColor(0, 1, 1);
create outline
(10) vtkOutlineFilter *outline = vtkOutlineFilter::New();
(11) outline->SetInput(sphereSource->GetOutput());
(12) vtkPolyDataMapper *outlineMapper = vtkPolyDataMapper::New();
(13) outlineMapper->SetInput(outline->GetOutput());
(14) vtkActor *outlineActor = vtkActor::New();
(15) outlineActor->SetMapper(outlineMapper);
(16) outlineActor->GetProperty()->SetColor(0,0,0);
```

Figure 10.11: An implementation in VTK
The code of Figure 10.13 animates the scene by managing time implicitly and using three loops that either alter the connections between abstractions or modify an abstraction’s attributes, such as color. The three loops partition the animation into three phases. In phase one, the animation renders the sphere ninety times (lines 18-20). Each rendering pass simulates the passage of one unit of time. In phase two, the bounding box connects with the renderer and the animation loops for twenty units so as to fade the bounding box from black to red (lines 21-25). After the loop concludes, the sphere disappears as it is disconnected from the renderer (line 26). In phase three, the animation loops for one hundred units to elevate the sphere and its bounding box for twenty spatial units (lines 27-45). For much of the loop, only the bounding box appears. The sphere appears by itself when the position of the bounding box falls within the range of five to ten units (line 31-37). The appearance and disappearance of the sphere and its bounding box are performed repeatedly by connecting and disconnecting the bindings to the renderer.

10.3.2 VRML’s Implementation

VRML (Virtual Reality Modeling Language) is an international standard for storing and sharing interactive 3D graphics. The current version VRML97 adds audio, video, and scripting capabilities to the hierarchical descriptions of static scenes. Although not explicitly designed for scientific visualization, VRML is gaining popularity in the visualization community as a means to develop visualization servers on the world wide web. Data and visualization software reside on servers that produce VRML files for visiting clients. VRML animates a scene by ordering nodes, establishing routes, and introducing scripts. Nodes are high-level abstractions that encapsulate shape, material attributes, and transformations. Every node retains a collection of fields that specify the node’s state. Routes are unidirectional links that

\[ \text{Vrml97 is the International Standard ISO/IEC 14772-1:1997.} \]
CHAPTER 10. IMPLEMENTING DOMAIN APPLICATIONS: A COMPARISON

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>int sphOn = 0; float pos[3]</td>
</tr>
<tr>
<td>18</td>
<td>animate from t=0 to t=90</td>
</tr>
<tr>
<td>19</td>
<td>for(int i=0; i&lt;90; i++)</td>
</tr>
<tr>
<td>20</td>
<td>renWin-&gt;Render();</td>
</tr>
<tr>
<td>21</td>
<td>animate from t=90 to t=120</td>
</tr>
<tr>
<td>22</td>
<td>for(int i=90; i&lt;120; i++)</td>
</tr>
<tr>
<td>23</td>
<td>outlineActor-&gt;GetProperty()-&gt;SetColor((i-90)/30.,0,0);</td>
</tr>
<tr>
<td>24</td>
<td>renWin-&gt;Render();</td>
</tr>
<tr>
<td>25</td>
<td>}</td>
</tr>
<tr>
<td>26</td>
<td>ren-&gt;RemoveActor(sphActor);</td>
</tr>
<tr>
<td>27</td>
<td>animate from t=100 to t=200</td>
</tr>
<tr>
<td>28</td>
<td>for(int i=100; i&lt;200; i++)</td>
</tr>
<tr>
<td>29</td>
<td>outlineActor-&gt;AddPosition( 0, 1, 0 );</td>
</tr>
<tr>
<td>30</td>
<td>sphActor-&gt;AddPosition( 0, 1, 0 );</td>
</tr>
<tr>
<td>31</td>
<td>pos = outlineActor-&gt;getPosition();</td>
</tr>
<tr>
<td>32</td>
<td>if (pos[1] &gt; 5 &amp;&amp; pos[1] &lt; 10) {</td>
</tr>
<tr>
<td>33</td>
<td>if (sphOn) {</td>
</tr>
<tr>
<td>34</td>
<td>ren-&gt;AddActor(sphActor);</td>
</tr>
<tr>
<td>35</td>
<td>ren-&gt;RemoveActor(outlineActor);</td>
</tr>
<tr>
<td>36</td>
<td>sphOn = 1;</td>
</tr>
<tr>
<td>37</td>
<td>}</td>
</tr>
<tr>
<td>38</td>
<td>else</td>
</tr>
<tr>
<td>39</td>
<td>if (sphOn) {</td>
</tr>
<tr>
<td>40</td>
<td>ren-&gt;RemoveActor(sphActor);</td>
</tr>
<tr>
<td>41</td>
<td>ren-&gt;AddActor(outlineActor);</td>
</tr>
<tr>
<td>42</td>
<td>sphOn = 0;</td>
</tr>
<tr>
<td>43</td>
<td>}</td>
</tr>
<tr>
<td>44</td>
<td>renWin-&gt;Render();</td>
</tr>
<tr>
<td>45</td>
<td>}</td>
</tr>
</tbody>
</table>

Figure 10.13: An implementation in Vtk

establish connections between the fields of nodes. Scripts are special nodes that encapsulate the functions of a scripting language, such as Javascript or Tcl. These functions determine how a VRML specification animates its state.

Building the Scene

The code of Figures 10.14, 10.15, and 10.16 implements the scene with VRML. The first two figures assemble a collection of nodes into a scene graph to produce a sphere, a bounding box,
and two interpolators, which ultimately update the position and color of the two shapes. The third figure produces a script and an assortment of routes. While the script controls the behaviors of the interpolators and the visibilities of the shapes, the routes communicate the change of state values between nodes.

The code of Figure 10.14 defines a sphere and its bounding box as shapes within a transformation, which determines the positions of the two shapes in world space (lines 2-23). With a Switch, a special node that selects one node from a group, the transformation also controls the visibilities of the two shapes, which appear together or as individuals (lines 7-21). The script of Figure 10.16 animates the visibilities of the shapes by updating whichChoice, the selector field for the Switch node (line 6). The prefix DEF, as defined with each shape, gives a node a name. A named node is accessed with the prefix USE, which the Switch node applies to integrate both shapes into a Group (line 17-19).

```
#VRML V2.0 utf8
DEF myTransform Transform {
  translation 0 0 0
  children [
    DEF mySwitch Switch {
      whichChoice 1
      choice [
        DEF myBOX Shape {
          appearance Appearance {
            material DEF myMaterial Material { diffuseColor 0 1 0 }
          }
          geometry Box { size 4 4 4 }
        }
        DEF mySPH Shape {
          geometry Sphere { radius 2 }
        }
        Group {
          children [ USE myBOX USE mySPH ]
        }
      ]
    }
  ]
}
```

Figure 10.14: An implementation in VRML

The code of Figure 10.15 defines two sensors and two interpolators, one for color and another for position. With the aid of routes, which appear in Figure 10.16, the two sensors

7In VRML, every node is defined globally, and all are implicitly part of a larger scene graph.
affect the computations of the interpolators. The sensors determine a time and a duration for which the interpolators compute their values. The duration set by a sensor is set by the field `cycleInterval`, which is specified in seconds. `myClockClr`, for instance, which has its duration set to ten, induces the color interpolator to produce a series of values over a period of ten seconds (line 26). The fields `startTime` and `stopTime` of each sensor indicate when a sensor begins and ends its execution over time. The values of both fields are set according to a real-time clock, which counts in seconds and starts at zero on midnight GMT, January 1, 1970.8

<table>
<thead>
<tr>
<th>creates a clock</th>
<th>creates a clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>(24) <code>DEF myClockClr TimeSensor {</code></td>
<td>(34) <code>DEF myClockMv TimeSensor {</code></td>
</tr>
<tr>
<td>(25) <code>loop FALSE</code></td>
<td>(35) <code>loop FALSE</code></td>
</tr>
<tr>
<td>(26) <code>cycleInterval 10.0</code></td>
<td>(36) <code>cycleInterval 100.0</code></td>
</tr>
<tr>
<td>(27) <code>startTime 0</code></td>
<td>(37) <code>startTime 0</code></td>
</tr>
<tr>
<td>(28) <code>stopTime 1</code></td>
<td>(38) <code>stopTime 1</code></td>
</tr>
<tr>
<td>(29) <code>}</code></td>
<td>(39) <code>}</code></td>
</tr>
<tr>
<td><code>creates a color interpolator</code></td>
<td><code>creates a color interpolator</code></td>
</tr>
<tr>
<td>(30) <code>DEF myColor ColorInterpolator {</code></td>
<td>(40) <code>DEF myMove PositionInterpolator {</code></td>
</tr>
<tr>
<td>(31) <code>key [0.0, 1.0]</code></td>
<td>(41) <code>key [0.0, 1.0]</code></td>
</tr>
<tr>
<td>(32) <code>keyValue [0 0 0, 1 0 0]</code></td>
<td>(42) <code>keyValue [0 0 0, 0 8 0]</code></td>
</tr>
<tr>
<td>(33) <code>}</code></td>
<td>(43) <code>}</code></td>
</tr>
</tbody>
</table>

Figure 10.15: An implementation in VRML

The code of Figure 10.16 defines a script node that applies Javascript (lines 44-67) and, as illustrated in Figure 10.17, an assortment of routes that connect the script node with the nodes of the scene graph (lines 68-76). The script node consists of five fields and three functions: `initialize`, `timerCheck` and `positionCheck`. In addition to storing state information, the fields support the functioning of the routes. When interacting with the routes, the fields are always public and are either unidirectional read or unidirectional write. The three functions work collectively to govern the actions of the interpolators. `initialize`, which is applied immediately when the node is read, computes a time ninety seconds into the future (lines 51-54). Forwarded by a route, that time determines when the color interpolator will begin to animate the color of the bounding box (line 68). `timerCheck` starts the positional interpolator after the color interpolator finishes, which is signaled when the interpolator's sensor `myClockClr` ends (lines 55-60, 71). While `timerCheck` waits to start the positional interpolator, the function sets the switch of the scene graph to render

---

8VRML will encounter a problem similar to Y2K when the values of its real-time clock rolls over in the distant future.
both the sphere and its bounding box (line 59). positionCheck examines the movements of the bounding box as set by the positional interpolator (lines 61-66). When the vertical position of the bounding box falls within the range of five to ten spatial units, the function alters the Switch node of the scene graph to render the sphere in place of the bounding box.

```plaintext
DEF myScript Script {
  unidirectional fields
  eventIn  SFBool  timerCheck
  eventIn  SFVec3f  positionCheck
  eventOut SFInt32  show
  eventOut SFTime  triggerTime
  eventOut SFTime  startT
  url "javascript:

  initialization during startup
  function initialize() {
    triggerTime = ((new Date()).getTime() / 1000.) + 90;
  }

  wait to start translation
  function timerCheck( value, time ) {
    if (value == FALSE)
      startT = time;
    else
      show = 2;
  }

  what renders during translation
  function positionCheck( value, time ) {
    if (value[1] > 5 && value[1] < 10)
      show = 1;
    else
      show = 0;
  }

  delay beginning of animation
  ROUTE myScript.triggerTime TO myClockClr.set_startTime

  animate color of bounding box
  ROUTE myClockClr.fraction_changed TO myColor.set_fraction

  start translation
  ROUTE myClockClr.isActive TO myScript.timerCheck

  ROUTE myScript.startT TO myClockMv.set_startTime

  translate shapes
  ROUTE myClockMv.fraction_changed TO myMove.set_fraction

  check movement shapes
  ROUTE myMove.value_changed TO myTransform.set_translation

  ROUTE myScript.positionCheck

Figure 10.16: An implementation in VRML
The routes of VRML operate by establishing unidirectional pathways between the fields of nodes, with one field as a source and another field as a receiver. When a source field changes state, the routes communicate the new state to the receiver field. The cascading effects of many routes form a chain of relations. For instance, in animating the position of the bounding box, three routes interconnect the script node to the shapes of the scene graph via a sensor and a positional interpolator (lines 72, 74-75). The first route, which communicates a state change from the script to the sensor, causes the second and third routes to communicate state changes from the sensor to the interpolator and from the interpolator to the transformation, which controls the position of the bounding box.

![Network Diagram](image)

Figure 10.17: The network produced by the routes of Figure 10.16

### 10.3.3 RASP’s Implementation

The code of Figures 10.18 and 10.19 implement the scene with RASP. The first figure builds a sphere, a bounding box, and a time graph while the second defines moveUp, a script for moving two shapes vertically. The time graph sequences moveUp and two additional scripts to coordinate the communication of information between components. The additional scripts, changeColor and renderModel, animate the color of a bounding box and the visibility of a shape. The definition of both appears in Figure A.9 of the appendix.

The code of Figure 10.18 creates a time graph with instances of shapes and scripts. The scripts forms connections between the shapes and other components, such as a renderer (lines 1-11). The time graph applies one interval and two interval modifiers to coordinate the execution of four different scripts: solidAct, outlineAct, colrAct, and moveAct (lines 12-18). The first script, solidAct, inherits an interval of time from 0 to 90. During that
time only the sphere’s state is communicated to the renderer. The second and third scripts work in conjunction with the first in inheriting an interval of 10 units from NextInterval, a modifier which starts an interval after the end of a previous one. From 90 to 100, the two shapes are visible and the color of the bounding box emerges gradually. The last script, moveAct, inherits its interval from a second instance of NextInterval. From time 100 to 200, the bounding box moves upwards with the sphere appearing whenever the vertical position of the box is greater than five and less than 10 spatial units.

```cpp
create sphere
(1) GeoSphere *sphere = new GeoSphere( 10 );
(2) Model *model = new Model( sphere );
create black bounding box (of sphere)
(3) Model *model2 = new Model( sphere->shallow() );
(4) Material *black = new Material( ColorBase::BASIC_BLACK );
(5) DrawingStyle *style = new DrawingStyle( DrawingStyle::BOUNDING_BOX );
(6) model2->setAttribute( black );
(7) model2->setAttribute( style );
form action units
(8) Activity *solidAct = renderModel( world, model );
(9) Activity *outlineAct = renderModel( world, model2 );
(10) Activity *colrAct = changeColor( model2 );
(11) Activity *moveAct = moveUp( world, model, model2 );
form time graph
(12) TmGroup *group = new TmGroup;
(13) group->addNode( new TmInterval(0,90) );
(14) group->addNode( solidAct );
(15) group->addNode( new NextInterval(10) );
(16) group->addNode( solidAct + outlineAct + clrAct );
(17) group->addNode( new NextInterval(100) );
(18) group->addNode( moveAct )
```

Figure 10.18: An implementation in RASP

The code of Figure 10.19 defines moveUp using one interpolator, six software interactions, and one virtual port. With the advancement of time, the interpolator and its associated software interactions interpolate a single value to move the positions of two shapes vertically (lines 19-21). As illustrated in the software network of Figure 10.20, a conditional event uses the virtual port to decide which of the two shapes will be communicated to the renderer (lines 26-31). The interaction setEvt applies a filter with its operations to perform a partial write on the position of the sphere and its bounding box (lines 22-25). The partial write communicates the vertical height of the shapes as a single value to inPosition, a port requiring a three-dimensional value. The remaining two dimensions are unchanged since
new values are not communicated to the port. The code concludes with an Activity that scripts the behavior of the interactions (lines 32-35). After the interpolator is provided with a duration, the script continually sequences through three interactions: updating the time of the interpolator, setting the positions of the shapes, and deciding which of the two shapes to render.

```cpp
Activity* moveUp( RaspSetting *world, Model *solid, Model *outline )
{
    // interpolator for moving models
    EvolveDbl *ev = new EvolveDbl( 0, 20 );
    DurationEvent *durEvt = new DurationEvent( ev->inDurationVal() );
    TimeEvent *tEvt = new TimeEvent( ev->inTime() );

    // partial write models’ position
    PtConnect *filter = new PtConnect( Point3::Y );
    Event *setEvt = new Event( ev->outCurVal() );
    setEvt->setFilter( filter );
    setEvt->addTarget( solid->inPosition(), outline->inPosition() );

    // communicate models ready for rendering
    Event *solEvt = new Event( solid->outThis(), world->inRender() );
    Event *outEvt = new Event( outline->outThis(), world->inRender() );

    // examines the state of the interpolator
    OPort *cond = *( *ev->outCurVal() > 5 ) && *( *ev->outCurVal() < 10 );

    // determines what model to render
    CondEvent *cEvt = new CondEvent( cond );
    cEvt->setTrueEvent( solEvt );
    cEvt->setFalseEvent( outEvt );

    // assemble interactions into an activity
    Activity *act = new Activity();
    act->addInitEvent( durEvt );
    act->addActEvent( tEvt, setEvt, cEvt );
    return act;
}
```

Figure 10.19: An implementation in RASP

10.3.4 Comparison

The implementation of VTK, VRML, and RASP are all similar in that each develops a scene from an assembly of parts and connections. VTK joins the inputs and outputs of components to form a demand-driven dataflow graph; VRML binds unidirectional fields with routes to assemble a network of pathways; and RASP interconnects unidirectional ports with software interactions to form a software network. The fundamental difference among the three tools is the approach applied by each to connect their respective parts. Only RASP provides mecha-
Figure 10.20: The software network produced by the code of Figure 10.19

nisms to vary the topology of its network of connections over time. For the sake of animating state, neither VTK nor VRML provides substantial means to vary the interconnections between its nodes (or components). A network with dynamic topology is useful for varying the presentation of complex data sets. For instance, a visualization may progressively advance through several phases in presenting a data set as an iso-surface, a scalar glyph, and a solid with cross-sectional cuts.

RASP is also the only tool of the three to provide explicit controls for managing simulation time and integrating simulation time with animating actions. In VTK, the advancement of simulation time is implicitly defined within the execution of its code. Modifying the timing of an animation is difficult, especially when parallelism is present. For instance, to separate the timing of the exemplary scene into three new intervals, such as those from 0 to 30, 31 to 60, and 61 to 90, the code of Figure 10.13 must re-define its current loops or introduce new loops with new instructions. Modifications to the timing of a scientific visualization is important in order to test and observe the behavior of a system in reaction to temporal changes. For instance, in the visualization of a weather system, continuous alterations to the timing of weather fronts are useful in reproducing a wide range of weather patterns.

In VRML, the timing of a visualization is controlled by a real-time clock. Script nodes and sensors reference real time values to disseminate temporal information to other nodes, such as interpolators. A real-time clock is useful for producing real-time visualizations, but not for producing visualizations that are computationally expensive or are estimates of long-term phenomena. With both types of visualizations, the nodes of VRML must apply numerical calculations, such as scaling, to adapt their timing values to the real-time clock.
For instance, should the rendering time for the sphere of the exemplary scene be costly, the timing of the remainder of the visualization should be adapted accordingly so as to convey a smoothly flowing presentation.

With its controls for managing time, RASP provides a structured approach to animating the contents of a visualization. A visualization develops from the building of high-level abstractions, such as scripts and meta-scripts, that plan the management and application of temporal values. This approach complements the dataflow approach of visualization which uses high-level functions to manipulate the values of data sets. RASP uses scripts and meta-scripts to manipulate the application of high-level functions. A structured approach also helps in presenting scientific data properly and in enabling the investigation of dynamic systems. A well-structured visualization accommodates design changes, such as those involving the visibility of its parts and the movements of cameras, to improve presentation. The script nodes of VRML do not offer the same capabilities as the scripts and meta-scripts of RASP because script nodes control only the production, not the dissemination, of information. The script nodes neither change the application of routes nor introduce a structured approach to managing the use of routes. Once they are set, the routes of VRML remain fixed and active.\footnote{VRML permits script nodes to add routes to a scene at run-time. These routes, however, are added haphazardly and are impossible to remove.}

### 10.4 Summary of Evaluation

This chapter has provided an empirical comparison between RASP and the representative tools of two disparate domains, algorithmic animation and scientific visualization. Algorithmic animation uses animation to present the behavior of computational algorithms while scientific visualization uses animation to analyze and present scientific data. The comparison performed for each domain involved the implementation of an exemplary scene with two tools, POLKA and OBLIQ-3D for algorithmic animation, and VTK and VRML for scientific visualization. Each of the tools provided insights into the development of animation for a specific domain and the desirability of various domain tools in producing animation.

The comparison demonstrates that IBS, as embodied by the RASP toolkit, supports the aims of both domains. For algorithmic animation, IBS supports a separation between an algorithm and the code that animates the behaviors of the algorithm: each part is developed separately so as to encourage reuse and to provide structure in the development of an algorithmic animation. For scientific visualization, IBS provides high-level abstractions for managing the development and execution of dataflow architectures. As demonstrated by
the visualization of a sphere and its bounding box, the abstractions manage the application of functions and data sets over time.

The comparison also shows that IBS and its support for animating methods, complexity measures, and software reuse are useful for building applications. Scripts and meta-scripts provide algorithmic animation with a uniform approach to animating state and precise controls for managing time. Components and objects need not be designed specifically to support one particular means of animation. Further, the timing of an animation easily accommodates the presentation of new behaviors and the rescheduling of animating actions, which may apply a variety of animating methods to animate state. The cylinders of the sorting algorithm, for instance, could easily be enhanced with multi-modeling methods to flash whenever their values are compared.

IBS provides scientific visualization with a means to vary the topologies of a dataflow architecture and to integrate these changes with the advancement of time. Connections between components (or nodes) vary either to alter the presentation of scientific data or to animate the state of a visualization that exhibits several phases. Scripts and meta-scripts are useful in structuring a visualization because the timing is easy to manage and the design is easy to control. The visualization of the sphere and its bounding box, for instance, can easily apply a new time graph to vary the times when the two shapes animate. Interaction constraints can also be introduced to limit the traveling speeds of the two shapes if they move too quickly, which would occur if the script controlling their movements were assigned a small interval of time.
Chapter 11

Easing Application Development: An Assessment

This chapter complements the previous two chapters by providing the third of three means of validating the design and application of IBS to visual simulation. By focusing on the management of scene complexity and support for software reuse, this chapter assesses the utility of IBS in easing the development of visual simulations. The assessment of IBS consists of logical arguments and empirical results. The logical arguments discuss the benefits of creating a visual simulation with dynamic topology and multi-modeling methods, showing how each helps to manage scene complexity. Dynamic topology effectively supports complex scene development with continually changing relations, and multi-modeling enables the development of complex scenes with iterative refinement. The empirical results section describes practical experiences using IBS and how the reuse guidelines improve the reuse of visual simulations developed independently and from legacy code.

11.1 Managing Complexity with Dynamic Topology

Programs with dynamic topology vary the interconnections that arise among software components. As the programs execute, rules and conditions determine when and how the interconnecting components communicate. Programs with dynamic topology are useful for visual simulations because they vary at run-time the times and conditions that determine when state values change.
11.1.1 Run-time Configuration

A program that progressively calls a sequence of functions could be perceived as changing its topology. However, the topology of the network, which the program represents, is set and determined by the ordering and execution of the function calls. Dramatic changes to the network altering its topology would be brought on by a re-ordering of its function calls; such a re-ordering would typically require the re-compilation of the program that applies the function calls.

For instance, the code in Figure 11.1 defines a program that sequences two functions, fool and foo2 (lines 1-8). Each of the two functions establishes a set of communications between four components: A, B, C, and D (line 9-11, 12-13). The diagrams of Figure 11.2a present the network configurations produced by fool and foo2. As the main program executes, the communications between components is fixed according to the program implementation, which sets an ordering of calls to fool and foo2. The program is unable to alter its run-time state to start foo2 before fool, and cannot operate both functions concurrently to replicate the network of Figure 11.2b, in which all four components communicate as a whole. The program is also unable to vary the application of the conditional that selectively calls fool or foo2 (line 4-7). The conditional operates repeatedly throughout the program's execution.

```c
void main()
{
    do {
        fool();
        foo2();
        if (< cond >)
            fool();
        else
            foo2();
    } while (< cond >);
}

void fool()
{
    B->set( A->get() );
    C->set( B->get() );
    A->set( C->get() );
}

void foo2()
{
    D->set( A->get() );
    D->set( C->get() );
}
```

Figure 11.1: The coding of a conventional program having fixed topology

In contrast, a program with dynamic topology alters its configuration with its execution. Its configuration is defined by several networks operating in progression. The ordering of networks, produced from scripts, is determined by run-time values and conditions, not by the ordering of function calls. Dramatic changes to a program's topology may be achieved by altering the execution of scripts, rather than by re-ordering a program's definition. With
the exclusion of scripts applied by another, scripts may be interpreted in any order and will still produce the same results, because it is the order of execution, not the order of interpretation, that ultimately determines when the networks take effect.

The pseudo-code of Figure 11.3 adapts the code of Figure 11.1 to differentiate the coding and development of a program with dynamic topology. Rather than having `foo1` and `foo2` explicitly produce the communications between components, these functions simply return two scripts, which the program stores as `s1` and `s2`. Each script uses software interactions to form networks similar to those of Figure 11.2a (lines 1-2). The conditional of the original program is produced as a script, referred to as `s3`, that selectively applies either `s1` or `s2` (lines 3). The topology of the original program is reproduced exactly with a sequential ordering of `s1`, `s2`, and `s3` (line 4). Small variations to the ordering of the three scripts alters the program's topology so that it no longer resembles that of the original program. A new ordering may have the networks of `foo1` and `foo2` operating in parallel (line 5) or in reverse order (line 6). Another ordering may have all three scripts operating at different times. The execution of `s1` and `s2` may overlap for a short interval of time while `s3` begins after a short delay (line 7). The overall timing of the original ordering or any of its variants may be set when the ordering is set to execute (line 8).

11.1.2 Application to Visual Simulations

Dynamic topology is beneficial to visual simulation for several reasons. First, it permits an application to freely vary the use of animated actions. Produced with scripts, the animated actions are not bound to operate according to the ordering of their definitions. For instance, all the animated actions of the previous two chapters may be freely applied without having to adapt their definitions or to evaluate them in the order of their intended execution. With the exclusion of obvious conflicts, such as two actions animating the same object, animated actions and their corresponding networks may also be freely combined to create cumulative
void main()
{
    s1 = foo1();
(1)
    s2 = foo2();
(2)
    s3 = CondScript(<cond>,s1,s2);
sequence scripts
(3)
    s4 = s1 >> s2 >> s3;
(4)
    s4a = s1 + s2 >> s3;
(5)
    s4b = s2 >> s1 >> s3;
(6)
    s4d = s1(0,15) & & s2(10,25) & & s3(30,40);
execute scripts
(7)
    execute s4(0,30)
(8)
}

Script foo1()
{
    Inter i1( A->get(), B->set() );
(9)
    Inter i2( B->get(), C->set() );
(10)
    Inter i3( C->get(), A->set() );
(11)
    return Script(i1,i2,i3);
(12)
}

Script foo2()
{
    Inter i4( A->get(), D->set() );
(13)
    Inter i5( C->get(), D->set() );
(14)
    return Script(i4,i5);
(15)
}

Figure 11.3: Pseudo-code that adapts the code of Figure 11.1 to have dynamic topology

effects. For example, the particle system animation of section 9.4 and procedural animation
of section 9.3 may be applied together so that the particles avoid the positions of the leaping
robot.

Second, dynamic topology permits a visual simulation to plan freely the timing of
animated actions. The animated actions and their corresponding networks are not confined
to execute sequentially. Over time, the actions may overlap, repeat, or encounter delays.
Figure 11.4, for instance, presents the timeline and active networks of an animation that
applies three scripts, all of which, for the sake of brevity, are greatly simplified in appearance.
Script#1 moves a robot towards a light (section 3.1.1), Script#2 instructs a particle system
to avoid collisions (section 9.4), and Script#3 moves a robot between two pucks (section 9.3).
As the animation executes, the particle system avoids the location of the moving robot,
which moves differently over time. The robot starts by moving toward a light and finishes
by leaping between two pucks. The overlap and the delays between scripts vary the topology
of the animation over time. During the times 20 to 30 and 45 to 55, the topologies of the
two scripts interact momentarily to form larger networks, as shown twice at the top of
Figure 11.4. The two larger networks develop and disappear dynamically with the passing
of time.

Third, dynamic topology permits visual simulations to produce animating actions
with a minimal set of components. Only components providing immediate use to an an­
imated action are networked into its definition. This is unlike that of many animation
systems, such as CORY [56] and SWAMP [16], that apply a fixed set of components to inter­
pret and process the descriptions of every animating action. Troubles arise quickly if the
set of components applied by the animation system is too small or too large. With too few components, the system is incapable of processing a wide variety of animating actions. Only actions reproducible with the small configuration of components are permissible. With too many components, the system becomes large, slow, and complex. To process effectively a wide variety of animating actions, the system must continually apply conditions to decide which of its components are useful. Only those components that cater to the needs of an animating action need be applied. If an animating action does not compute collisions between objects, for instance, the system should not apply components and functions that perform such tasks.

Finally, dynamic topology enables visual simulations to partition any given script with an equivalent ordering of smaller scripts so as to add variability to the use of an animating action. The network of the original script is reproduced exactly with an appropriate ordering of the networks of the smaller scripts. Alterations to this ordering, such as those changing the times, durations, and conditions of specific interactions, adjust the behavior of the original script. For instance, the illustration of Figure 11.5 rearranges the network of a larger script into two smaller ones. The larger network, which controls the movement of a robot between two pucks (section 9.3), is divided into a network that provides an interpolator with the positional value of a puck, and another network that applies the same interpolator to animate the movement of the robot. Rather than having the two small networks operate strictly in parallel, which would replicate the workings of the original network, the second network is set with a longer duration and greater granularity. From time 50 to 70, the puck communicates its position continuously to the interpolator, which intermittently sends the robot positional values. During this time, the robot moves irregularly, as though it were illuminated with a strobe. From time 70 to 80, only the robot continues its communications.
with the interpolator. During that time, the robot moves towards a location that the puck had attained at time 70.

![Figure 11.5: The partitioning of a software network into several smaller ones](image)

11.2 Managing Complexity with Multi-Modeling

Multi-modeling methods are useful for building visual simulations of several parts. Each part of a visual simulation is produced with the method that best matches its design and nature. These methods work collaboratively to form a single working application. Multi-modeling methods are also useful for expanding and augmenting visual simulations with new parts. As a visual simulation expands, it may grow by progressively adding new methods, not necessarily by subjugating its existing methods to accommodate the workings of new parts. The existing methods simply replace or augment its current set of interactions with newer ones. The newer interactions refine or enhance the network of communications that the visual simulation applies to animate state.

11.2.1 Application to Visual Simulation

The application of multi-modeling to visual simulation is important because it enables scenes to develop incrementally without needing to undergo significant adjustment to accommodate greater detail. A scene may begin simply as a small set of components and interactions, and then grow gradually in size as the interactions and components are replaced with newer ones of greater complexity. For instance, a single interaction may be replaced with a declarative event that refines the communications of the original with a finite state automata.

The utility of multi-modeling to visual simulation is apparent in the example of Figure 11.6, which expands the scene of Figure 9.8. The diagram of Figure 11.6a illustrates the execution of the original script while the image of Figure 11.6b presents a visual representation of the revised scene, which adds the movements of a rocket to the movements
of a the leaping robot. Each time the robot advances towards the second puck, the rocket shoots upwards from the current position of the robot. When the robot begins its return to the first puck, the rocket halts, flashes ten times, and waits to begin again.

Divided into four parts, the code in Figure 11.7 presents a set of interactions for detailing the motions of the rocket. These interactions will be applied in updating the original scene with a functional modeling method. In the first part, three interactions communicate information among the rocket, the robot, and the interpolator (line 23-25). \texttt{bgnSet} sets the interpolator’s initial value to equal the robot’s position; \texttt{tEvtRck} updates the interpolator with temporal information; and \texttt{setRck} communicates a positional value from the interpolator to the rocket. Collectively, the three interactions enable the interpolator to move the rocket skyward. In the second part, the virtual event \texttt{stateMv} determines whether the robot actively advances towards the position of the second puck (line 26). The virtual event ascertains this information by expressing a logical relation with \texttt{ef2}, the event that advances the robot’s position. Upon receiving a logical request from either \texttt{waitEvent} or \texttt{whileEvent}, \texttt{stateMv} returns true if it executes in the same time step with \texttt{ef2} (line 27-28). In the third part, \texttt{whileEvt} updates the position of the rocket if the robot is moving towards the second puck. When applied by a script, both \texttt{whileEvt} and \texttt{waitEvt} halt the execution of further interactions until each obtains a false condition from \texttt{stateMv}. The difference between the two events is that \texttt{whileEvt} not only suspends as it awaits a false condition, but also executes \texttt{tEvtRck} and \texttt{setRck}, the events for advancing the rocket’s position (line 29). In the final part, two interactions, \texttt{tglEvt} and \texttt{loopEvt}, set the rocket to flash ten times (line 30-33). \texttt{tglEvt} handles the flashing of the light, and \texttt{loopEvt} enables the repetition.

The code in Figure 11.8 applies the interactions of Figure 11.7 to augment \texttt{act}, the activity of the original script (see Figures 9.13) (line 33). The new version of \texttt{act} sequences
CHAPTER 11. EASING APPLICATION DEVELOPMENT: AN ASSESSMENT

... interpolator to move rocket
(22) EvolutePt *ev = new EvolutePt;
communicate to and from the interpolator
(23) Event *bgnSet = new Event( robot->outPosition(), ev->inBeginVal() );
(24) TimeEvent *tEvtRck = new TimeEvent( ev->inCalcValue() );
(25) Event *setRck = new Event( ev->outCurVal(), rocket->inPosition() );
 hold if robot is moving to puck2
(26) VEvent *stateMv = VEvent( *ef2 );
(27) WaitEvent *waitEvt = new WaitEvent( stateMv );
 move rocket if robot is moving towards pucks2
(28) WhileEvent *whileEvt = new WhileEvent( stateMv );
(29) whileEvt->addEvent( tEvtRck, setRck );
 flash the rocket 10 times
(30) ToggleEvent *tglEvt = new ToggleEvent( rocket->inDrawBool() );
(31) tglEvt->setToggleValues( TRUE, FALSE );
(32) LoopEvent *loopEvt = new LoopEvent( 10 );
(33) loopEvt->addEvent( tglEvt );
...

Figure 11.7: Interactions for moving the rocket and having the rocket flash ten times

two sets of interactions concurrently. The first sequence SEQ1 replicates the interactions of
the original script for controlling the robot. As time flows, eSet follows the interactions of
the active state of fsm (line 34). The second sequence SEQ2 introduces a functional method
for controlling the interactions of the rocket. Unlike the first sequence, the second does not
repeat continuously for each time step. The sequence begins by halting until the rocket is
ready to move. Once the rocket is ready, bgnEvt initializes the rocket’s interpolator and
whileEvt repeatedly updates the rocket’s position until its enabling condition fails. loopEvt
completes the sequence by flashing the rocket ten times.

... establish script w/two sequences
(33) MultiActivity *act = new MultiActivity;
(34) act->addActEvent( SEQ1, fsa, eSet );
(35) act->addActEvent( SEQ2, waitEvt, bgnEvt, whileEvt, loopEvt );
...

Figure 11.8: The script of Figure 9.13 augmented with the interactions of Figure 11.7

The diagram in Figure 11.9 illustrates the execution of the script of Figure 11.8. Each
sequence of the script executes in parallel. Parallelism is achieved by employing separate
threads or by alternating a single thread between both sequences. At every time step, the
script executes each sequence completely. For SEQ1, \( \text{fsa} \) and \( \text{eSet} \) execute in tandem. For SEQ2, its interactions execute progressively. Both \( \text{waitEvt} \) and \( \text{whileEvt} \) may suspend the sequence from executing further interactions. The stippled arrows of Figure 11.9 signify the effects of suspension. If \( \text{whileEvt} \) suspends, \( \text{loopEvt} \) is ignored and the script unit repeats. Upon return, the sequence begins from \( \text{whileEvt} \). \( \text{loopEvt} \) follows when \( \text{whileEvt} \) finishes, which may not occur for several time steps. Once executed, \( \text{loopEvt} \) performs similarly to \( \text{whileEvt} \). It suspends, but only for a fixed number of repetitions.

![Diagram of script interactions](image)

**Figure 11.9:** An update to the script of Figure 11.6a that repeatedly executes two sets of interactions concurrently.

### 11.3 Supporting Reuse with the Guidelines

The guidelines of chapter 8 are intended to improve the reuse of software components. These properties ensure that components interact properly, behave similarly, and operate effectively with time. Preliminary experience with the guidelines indicates that the properties are effective in practice. The experiences consist of two approaches: *production* and *adaptation.*

**Production** involves the development of new applications with IBS serving as their foundation. Each application and its associated set of components were produced independently using software interactions as the means to coordinate software communications. **Adaptation** involves the re-engineering of existing applications into interaction-based programs. The new programs update the designs of old applications — programs built without RASP and usually without the use of any abstractions for communications — to support software interactions and to adhere to the design principles of IBS.
11.3.1 Production

A wide range of applications demonstrating the utility of IBS appear on the RASP home page. As can be noted from the images of Figure 11.10, some applications apply key-frame animation techniques, and others employ differential techniques and adaptive time steps to produce physically-based motions. The complexity of scenes within the applications range from the fairly simple to the rather complex.

![Images of applications](URL: www.cs.ubc.ca/nest/imager/contributions/gslee/Rasp/rasp.html)

Figure 11.10: Samples of applications built with RASP

Despite the dissimilarities among the applications, several commonalities emerged from the development of each. First, most coding involved the specification and creation of visual elements. The code for scene development exceeded that of scene simulation, except for long-running applications with many intricate changes. Second, the development of visual elements consumed less time than did the creation of a network of interactions followed by the assembly of the interactions into scripts. Although the latter two steps involved less code, they required more time to develop. There are fewer conventions in scene simulation than there are in scene development. Thus, it takes longer to describe an application by the way it changes state than by the way it controls its visual appearance. Third, the time to fine-tune an application was great, but not excessive. Precise changes to the appearance or behavior of an application required great care. The production of visual imagery is an art form, not an exact science.

Reuse

Using IBS, it was possible to reuse the components of prior applications. The data from Table 11.1 presents a breakdown of the number of components, scripts, and interactions.

---

1 URL: www.cs.ubc.ca/nest/imager/contributions/gslee/Rasp/rasp.html
2 The images of Figure 11.10 were developed with RASP's interface to Rayshade [82], a public domain ray-tracer.
of various applications. The information in the first two columns provides an overview of the number of distinct components in a simulation compared to the total number of components, showing that many of the components were reused across several applications. Some examples of component reuse are the component RobotArm, an articulated appendage that appears in four simulations, the component EvolvePt, a numerical interpolant that appears in every simulation, and the geometric shapes GeoCylinder and GeoCube, which appear repeatedly throughout.

<table>
<thead>
<tr>
<th>Title</th>
<th>Components</th>
<th>Scripts</th>
<th>Distinct # of Interactions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Distinct</td>
<td>Total</td>
<td>Distinct</td>
</tr>
<tr>
<td>Catch &gt; •</td>
<td>8</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>ConeTree &gt;</td>
<td>9</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>Flock &gt; •</td>
<td>7</td>
<td>47</td>
<td>3</td>
</tr>
<tr>
<td>Bang &gt; •</td>
<td>5</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>Stream &gt; •</td>
<td>9</td>
<td>68</td>
<td>6</td>
</tr>
<tr>
<td>Flyby &gt; •</td>
<td>9</td>
<td>57</td>
<td>6</td>
</tr>
<tr>
<td>Fountain &gt; •</td>
<td>8</td>
<td>210</td>
<td>4</td>
</tr>
<tr>
<td>Galaxy &gt; •</td>
<td>5</td>
<td>306</td>
<td>5</td>
</tr>
<tr>
<td>Handoff &gt; • •</td>
<td>8</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>Stampede &gt; •</td>
<td>7</td>
<td>29</td>
<td>3</td>
</tr>
<tr>
<td>Rigid &gt; •</td>
<td>7</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>Wave &gt;</td>
<td>8</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>Robots &gt; • •</td>
<td>7</td>
<td>13</td>
<td>5</td>
</tr>
<tr>
<td>Clouds &gt;</td>
<td>8</td>
<td>164</td>
<td>2</td>
</tr>
<tr>
<td>Rolling &gt; •</td>
<td>8</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>Smash &gt;</td>
<td>8</td>
<td>66</td>
<td>7</td>
</tr>
<tr>
<td>Solar &gt;</td>
<td>14</td>
<td>259</td>
<td>5</td>
</tr>
<tr>
<td>Sort &gt; •</td>
<td>7</td>
<td>47</td>
<td>5</td>
</tr>
<tr>
<td>Spin &gt; •</td>
<td>8</td>
<td>696</td>
<td>4</td>
</tr>
<tr>
<td>Compress &gt;</td>
<td>10</td>
<td>53</td>
<td>6</td>
</tr>
<tr>
<td>LiftUp &gt; • •</td>
<td>10</td>
<td>10760</td>
<td>3</td>
</tr>
</tbody>
</table>

Simulation contains one or more:

- EvolvePt
- RobotArm
- GeoCylinder or GeoCube

Table 11.1: Various simulations produced with RASP

The reuse of various components within and across applications was accomplished without significant interface or behavioral modifications to the components. Although definitive evidence is unavailable, this reuse is likely due to the design of IBS and its amenities, such as the guidelines for component design (see chapter 8). For example, it was possible to reuse the component RobotArm because it was programmed to respond based on its current state (PROPERTY B–2). In the application “handoff,” the robot is always able to provide the current position of its hand immediately after receiving an instruction from any script to update the angle of its base joint. In the different reuse contexts, the RobotArm was
manipulated by several interactions of several scripts. Had this component not always based its responses on recent inputs, its interactions would have fed other components with false values. The \texttt{EvolvePt} component, on the other hand, was reusable because it was coded to be time-invariant (\texttt{PROPERTY T-1}) and to respond to $\Delta T$ (\texttt{PROPERTY T-3}). These features allowed interactions to use the component in different time contexts and to synchronize appropriately the component's behavior over time. In applications "Rigid" and "Flyby," for example, \texttt{EvolvePt} operated with widely differing time steps.

### 11.3.2 Adaptation

As shown in Figure 11.11, IBS and its guidelines were applied to the building of two simulations that incorporate existing C++ components. Derived from standard programming practices, the components were developed independent of RASP. The "stampede" simulation animates a herd of wild horses [161]; the "mesh compress" simulation visualizes an algorithm for compressing geometrical datasets (see Isenburg [70] for details). In each case, the existing components were not developed in an interaction-based style. There was little or no separation between the computational tasks and the mechanisms controlling communication.

![Figure 11.11: Adapted applications with RASP](image)

(a) Stampede | (b) Mesh Compress$^4$

As shown in Table 11.2, the original components violated a number of properties that characterize a reusable component in the interaction-based approach. To remedy that situation, I extracted and used only the computational elements. These extracted components were then prepared by adding first-class operations, separating multi-parameter methods into elementary sets of unary operations, and by providing explicit support for time.

With their adaptations, the final versions of "stampede" and "mesh compress" became more flexible and more reusable than their original forms. They became easier to

---

$^4$Courtesy of Martin Isenburg
adapt for further uses with other components in new contexts. For example, the ambulatory motions of stampede adapt well to control human figure movements in a new simulation of walking robots while the compression algorithm of “mesh compress” can be used to produce view dependent images where the shapes of objects vary according to the current viewing parameters.

### 11.3.3 Preliminary Experiences

Preliminary experiences with RASP demonstrate that it is not difficult to program according to the guidelines. Although these experiences are limited to only a few individuals, those who have used the guidelines have been able to learn and apply them relatively easily. The guidelines that require the most effort to apply are interactional property I-3 and behavioral property B-2.

To follow property I-3, the general interface property, programmers reorganize the computations and interfaces of their components. They redesign the interfaces to support operations that are unidirectional read, unidirectional write, or bidirectional write. Bidirectional read operations, which require interactions to provide information before reading occurs, are eliminated by splitting them into two separate operations, one unidirectional write and one unidirectional read. The unidirectional write operation accepts information in the same format as the original bidirectional operation and then caches its output for the unidirectional read operation to provide as outgoing data.

Of the three accepted types of operations, only bidirectional write operations require additional work to implement. Programmers must determine whether it is possible for candidate operations to provide suitable data for their calling interactions to use before writing occurs. For most operations, this determination is not easy. It can often be difficult to determine what kind of information the interactions should receive from the operations. Once bidirectional write operations are created, programmers must also decide if the operations can or should support partial writes. Operations that accept and set multi-field data
typically adapt well. For example, operations manipulating three-dimensional values can typically be modified to work properly in fewer dimensions. When programmers are unable to form bidirectional write operations, they may divide the operations into their constituent unidirectional counterparts. For instance, in several of the animations from Table 11.1, it was not practical to provide bidirectional write operations for components computing movements from a combination of physical forces, such as velocity and acceleration. Since the data structures of these operations referenced several vectors and scalar quantities, it was not always clear which structures were to be set with partial data. It was best for the write operations to accept only complete values.

To meet property B-2, the elementary set property, programmers divide their multi-parameter operations into many unidirectionals. For many candidate operations, this task can be difficult to achieve because of corollary B-2, the side-effects property. Some multi-parameter operations, such as those that maintain records of their input, may not easily divide into several unary operations. For these types of operations, it is necessary to build additional operations that either induce the side-effect separately, or that accept all the arguments within a multi-field data structure. The latter is the method of choice because it adheres to property B-1, the current state property. For instance, in the “Rolling” animation of Table 11.1, the multi-parameter operations of a component for solving geometric constraints did not partition well into several unary operations. Side-effects would vary according to the execution order of the operations. To resolve this situation, a separate operation was introduced for signaling side-effects and for combining the parameter values provided to previously executing operations. This adaptation requires that all interactions affecting the component also use the signaling operation.
Chapter 12

Related Work

The literature presents many approaches to software development that apply interconnecting mechanisms between software components. Similar to software interactions, these mechanisms carefully monitor and control the information passing between the interfaces of components. Information flows properly if communications between components are of the right type, occur at the right time, and are semantically correct. Software interactions differ from the mechanisms of previous approaches in that software interactions are generic, dynamic, and organizable. As discussed in the previous chapters, software interactions work well with all kinds of components, produce systems with actively changing topology, and provide assistance in managing software complexity and improving software reuse.

This chapter discusses current approaches in software engineering and visual simulation that develop software with interconnecting mechanisms. It details the applicability of these approaches to support the development of visual simulations and to improve the development of reusable code. It also provides a commentary on dataflow networks, which are similar yet structurally different from software networks, and which develop from the use of software interactions.

12.1 Interconnections in Software Engineering

The interconnection of components with software interactions share many commonalities with semantic data modeling, connectivity modeling, software architecture design, and module interconnection languages. All three methods distinguish between elements that compute and elements that interconnect.
12.1.1 Semantic Data Modeling

Semantic data modeling methods, such as the entity-relationship (ER) data modeling methods of Chen [28] and Rumbaugh [129], employ "relations" to identify the associations (and constraints) among independently computing components and processes. The relations externalize the associations between components such that the associations merit their own notation and predefined operations. Although similar to software interactions, the relations of ER data modeling methods are typically second-class, specific to individual applications, and not ordinarily adaptable. Thus, they are difficult to vary over time and to use in alternative contexts. Moreover, the relations are unlike software interactions in that the relations do not address concurrency or synchronization issues – the critical issues of visual simulation. It is impossible to use ER data modeling methods to schedule multiple communications that perform in parallel or in sequence.

12.1.2 Module Interconnection Languages

Module interconnection languages (MILs), such as Mil-75 [35], provide a formal notation for identifying software components and specifying the requirements of component interconnections. The MILs state how the components of a system fit together and what the components expect and provide from their communications. Typically, an automatic process analyzes and verifies the integrity of a system by examining its interconnections, as indicated by an MIL specification. Generally, this process is neither dynamic nor adaptable. MILs also advocate specific structural designs. They assume particular forms of communications and do not differentiate what systems they describe or how components implement their tasks [121]. Interconnections between components are generally described with "uses" relationships and supported with only one kind of connection. Existing MILs are not suited for managing visual simulations because they neither incorporate nor address dynamic interconnections. Similar to ADLs, MILs lack features in their specifications for managing time and for managing numerous interconnections collectively.

12.1.3 Architecture Descriptions

Architecture descriptions and the languages that implement them (ADLs) model programs in terms of high-level abstractions, such as components and interconnections [115, 55]. Independent of any programming language, they permit developers to design formally large-scale software systems without immediately addressing or implementing low-level concerns, such as the organization of function calls or the definition of intricate data structures. Much of
the work with ADLs revolve around static analysis and system generation. They seek to increase the understandability of architectural designs and to encourage tool support for architecture-based development and evolution [100].

In bridging the gap from design to implementation, ADLs support a wide variety of mechanisms to specify and develop the interconnections between components. These mechanisms, none of which are standard, are usually modeled abstractly, explicitly, or inline. SYNOPSIS [34] and WRIGHT [9] model their interconnections abstractly with special descriptors. SYNOPSIS uses specification-level abstractions that encode interconnecting relationships, while WRIGHT applies a formal notation for ensuring that interacting components adhere to specific protocols. UNICON [139]\(^1\) models its interconnections explicitly as first-class abstractions. Their connectors regulate data flow and resource allocation. These ADLs employ port-like structures and advanced compilers in ensuring the genericity of their connections and the validity of communications between components. DARWIN [93] and RAPIDE [91] model their interconnections in-line with the specification of components. Rarely appearing as first-class or discrete entities, these connectors simply denote the connections that interconnecting components implement.

Although architecture descriptions are useful in specifying interconnections, they require significant effort to use in developing visual simulations. First, it is not completely clear how programs develop concretely from the use of an ADL. The refinement of an architecture description into source code is an open problem with many unanswered questions. For example, the ambiguity of some architectures allow multiple interpretations, some of which are inaccurate or unintended [103]. Second, basic architecture descriptions neither vary their interconnections dynamically nor assemble their interconnections into groups. Hence, the architecture descriptions are not well suited for sequencing or planning the use of interconnections over time.

### 12.1.4 Dynamic Software Architectures

Dynamic software architectures (DSAs) describe software systems that change topologies and their controls during system execution. Topological changes are produced with the introduction of new components, the removal of old components, and the re-configuration of the mapping of software to hardware. Control changes are produced with the creation and destruction of processes, and the suspension and resumption of threads.

DSAs vary widely in their designs and their means of enabling topological change.\(^1\)

\(^1\)In UNICON, connectors are conceptually discrete entities, but they are not implemented as such. The connectors may be reified, for example, as linker instructions during system generation.
Some DSAs, such as Rapide [91] and Dynamic Wright [8], extend ADLs with dynamic properties. These DSAs introduce explicit languages and notations for describing dynamically evolving software architectures. In general, these DSAs are limited in the kinds and types of topological changes they may produce since all the changes to a system's topology must be specified or compiled into the definition of a system. Other DSAs, such as C2's AML [99] and Clipper [3], strive for a more general approach by defining architectural modification languages (AMLs). AMLs establish a specification for identifying architectural changes. For large systems and those that are long-running or mission critical, AMLs facilitate extensions, customizations, and evolutionary changes.

Although DSAs are dynamic, they are not yet suitable for developing visual simulations. Their purposes are different in that they seek to change a system's definition and behavior at a relatively high level of abstraction. The changes to a system's architecture must be analyzed to ensure that the changes are proper and do not radically impact the on-going computations of a program. In addition, DSAs do not provide the utilities for developing visual simulations. They neither possess notions of time nor engage in the management of parallel state changes. Furthermore, it is unclear how their abstractions, like those of ADLs, are refined to produce source code and how their abstractions decide when and how dynamic state changes occur.

12.1.5 Connectivity Modeling

Connectivity modeling methods, such as Mediators [147], Flo [37], ACT [5], and Conduit [173], employ first-class connectors to coordinate interactions between components and processes. Developed to meet the needs of specific applications, the connectors of Mediators realize behavioral relationships between components so as to anticipate evolutionary changes to the integration requirements of a program. The connectors realize behavioral relationships by invoking functions in response to events, which components raise as they either change state or enact a specific action. The connectors of Flo are dynamic, user-defined abstractions that regulate sequential communications and enforce state changes in components. The connectors of Mediators and Flo are unlike software interactions in that they are arbitrarily complex. Each may contain algorithmic code to ensure that components apply specific patterns and protocols of communications. Hence, they are not as reusable in many contexts as software interactions are. In addition, the connectors of Mediators and Flo provide few means to observe and manage their actions with external controls, such as that of a meta-script. Hence, they limit the way that a visual simulation may animate. To express behavioral associations between components, such as a trigger, the connectors inter-
mix control statements with logical expressions. The logical expressions induce state changes that are difficult to observe and to coordinate over time. Finally, the connectors of Mediators and Flo are independent entities that rely upon their own agendas in interconnecting software components. They neither take cues nor accept commands to regulate their actions. Thus, the connectors are difficult to apply in coordinating sequences of interconnections over time.

In contrast, the connectors of ACT and Conduit bear some resemblance to software interactions. The connectors of both modeling methods are simple and easy to organize, and each serves as a primitive mechanism to construct abstractions for collaborative communications. The connectors of ACT operate with filters to intercept the data that enter and leave the interfaces of components. In intercepting the data, the connectors enact invariant behaviors, such as enforcing constraints or coordinating atomic transactions. The connectors of Conduit form a framework for developing communications protocols. The connectors join together to form pathways between communicating components. Despite their similarities to software interactions, the connectors of ACT and Conduit are applied differently than are software interactions. The connectors of ACT require components to acknowledge their presence in providing them with information. Thus, the connectors operate at the expense of component reuse. Components must remove and update their references to all connectors as the components become involved in new communications. The connectors of Conduit are messengers, as opposed to controllers. They receive information and forward the information to other components. This approach is useful for controlling communications between layered systems, but it is not amenable for systems with dynamically changing topologies. The approach prevents external influences from determining when, why, and how the components communicate. Moreover, the approach is not well suited for synchronizing communications and controlling concurrency, both of which are critical for visual simulation.

12.2 Interconnections in Visual Simulation

The use of interconnections to join software components has also been applied actively to the development of visual simulation. Many methods of computer graphics and computer simulation apply interconnecting mechanisms either to produce computational algorithms or to construct dataflow networks from computational components. Unlike software interactions, these mechanisms are applied primarily to express links between computational structures, not to set the configuration of a system and coordinate its communications.
12.2.1 Computer Graphics

In computer graphics, the use of interconnecting mechanisms has been applied actively to develop computations, animations, and scientific visualizations. In developing computations, the mechanisms form constraint graphs, an arrangement of components bound with numerical constraints. Similar to interaction constraints, which coordinate communications in IBS, constraint graphs create dependencies and simplify the production of computational algorithms. In developing animations and scientific visualizations, the mechanisms form dataflow networks, networks that identify the transport of information between components. As information moves through the network, each component uses the information it receives to either animate state or produce visuals.

Computation

In building constraint graphs, tools such as CONMAN [63], CONDOR [76], and BRAMBLE [58], interconnect components to express numerical functions and formulas. CONDOR and BRAMBLE apply constraint graphs to simplify the use of advanced numerical modeling techniques, such as interval arithmetic and differential evaluation, to the problems of computer graphics. With CONDOR, constraint graphs are interactively composed and manipulated to express complex mathematical forms, such as those for creating shaders, surfaces, and constrained models. With BRAMBLE, constraint graphs are interactively assembled to constrain and to control the attributes of graphical objects, such as their location and orientation. CONMAN applies a constraint graph to provide a visual programming environment. The constraint graph is produced interactively so as to provide a graphical facility for connecting visually-oriented tools, such as color interpolants and curve editors.

Although useful, these tools and their corresponding constraint graphs are not immediately adaptable for animating visual simulations. None of the tools provides effective means for varying the interconnections of a constraint graph so as to produce animating actions. Without dynamic controls, visual simulations are produced haphazardly with an assortment of conditionals, which selectively determine when and how state changes occur. The production is unsystematic since the development of animated actions is unguided by any rules or mechanisms that regulate the development and application of animating actions, which undoubtedly may be produced with changes to the constraint network. In addition, none of the tools provides explicit support for linking animating actions with the advancement of simulation time. To animate with time, developers must introduce their own methods, which may eventually lead to difficulties in reuse if the developers apply in-
consistent approaches across various applications. In Bramble, the only tool of the three to possess any notion of time, simulation time is integrated directly into the computations of a constraint graph. Time is applied to guide computations, not to guide the use of animating actions over time.

**Animation**

It is uncommon for the tools of computer graphics to apply interconnections directly to the task of animating visual simulations. Of the few tools that do, such as Frames [118], VRML [18] and Houdini [140], these tools apply interconnections to link components with time-varying structures, with an interpolant being the most common abstraction. As time advances, either the structures update the components with new state information or the components provide the structures with information to animate. These structures are useful for building prototypes, not for improving the design of visual simulations. They suffer the same drawbacks as those associated with the mechanisms that build constraint graphs. Neither Frames, which employs UNIX pipes [79] to pass information between executable filters, nor VRML, which employs routes to interconnect (the fields of) components, provides a systematic approach to forming animating actions from an assembly of interconnections. Animation arises simply from the workings of a global set of interconnections. The script nodes of VRML, which assist in developing visual simulations, are useful for affecting the communications between components, but not for varying the use of routes in interconnecting components. The interconnections of Houdini assemble a visual procedural network, a dataflow architecture with a visual representation. The network assembles high-level primitives into a procedural definition. Like dataflow architectures (see section 12.3), the interconnections of Houdini are neither first-class nor dynamic. Animation derives from the changing state of primitives, and the pushing and pulling of information throughout the network.

**Scientific Visualization**

Many toolkits for scientific visualization, such as VTK [134] and Iris Explorer [49], are presented and developed as modular visualization environments (MVEs). These toolkits facilitate the development of visual simulations by providing various interconnecting mechanisms for controlling the flow of information, which are usually large data sets, between computational abstractions. The interconnecting mechanisms are applied to the abstractions with the aid of an interactive visual interface and a specialized grammar for spatial presentations. The abstractions perform an assortment of operations to visualize the attributes and
behaviors of time-varying data sets. Common operations include the use of transformations, filters, and mapping techniques.

In interconnecting the abstractions, the interconnecting mechanisms produce a dataflow network, as presented and discussed in section 12.3. Hence, MVEs suffer the same drawbacks to visual simulation that dataflow networks do: topological changes are not generally possible, and the management of time is not easily achieved. MVEs normally support animation either by adding time stamps to data sets or by interpreting script files. The time stamps associate the states of data sets with future times, and the scripts instruct the MVEs to introduce changes to the data sets or the configuration of interconnections. However, like many tools for visual simulation that rely on scripts (such as IRIS EXPLORER, which uses SCHEME), the MVEs provide few rules or guidelines for building and managing the scripts methodically. Thus, the scripts are often difficult to produce in a manner that would enable them consistently to ease the management of complexity and support the application of software reuse.

12.2.2 Computer Simulation

Most simulation tools that interconnect components resemble those of scientific visualization. Simulation tools interconnect components either to form systems that resemble dataflow networks or directed graphs, or to join components in producing coupled models, systems designed from the intermixing of multiple modeling methods. These tools are generally associated with functional approaches (section 2.2.2) and are best for solving systems of continuous equations or for moderating systems with a directionality of flow between components, such as queues and control engineering. Many of these tools provide few mechanisms to alter the topology of the connections between components. Topological changes are often awkward and sometimes impossible to specify. For example, SES/WORKSBENCH [138] and PRISM [156] apply logical conditions to the edges (connectors) of their directed graphs. Components and connections are always present and accounted for by the simulator. They do not appear and disappear dynamically, as would happen in a truly dynamic network. DYMOLA [43] uses second-class connectors either to join or to relate the terminals (second-class ports) of components. Except for a few operations, such as path reversal and looping, the connections among terminals are fixed.

Two notable exceptions from computer simulation that support dynamic connectivity are DOSE [95] and SIMII [44]. Each employs a special module called a connection manager to form interdependencies between communicating components. As a simulation executes, the connection manager, which maintains a list of connections, controls the appearance and
disappearance of interdependencies. In DOSE, the connections emulate the links of dataflow networks, like those formed within modular visualization environments (section 12.2.1). The connection manager updates the links by scheduling events that signal state changes. As events take place, the state changes initiate communications. In SimII, the connections emulate the links of constraint graphs, similar to those developed with the tools of computer graphics. The connection manager evaluates the links by converting them into mathematical equations. Each time the links change, the equations are updated and re-solved.

As noted by several simulationists [44], the implementation of dynamic connectivity introduces several benefits to the development of simulations, regardless of whether they are visual simulations. First, it promotes software reuse. The logical behavior of components are developed separately from their bindings. Thus, the components are easier to comprehend and to extract. Second, it encourages the development of semi-reflexive systems that adapt to their own operations. Systems may wish to adapt their configurations to resolve operational considerations, such as real-time constraints or resource limitations. Third, it separates components from their experimental frames. As noted by Ziegler [170], an experimental frame defines the environment in which a simulation’s components operate. Components produced separately from their experimental frames are easier to validate because they test easily in multiple environments.

12.3 Interconnections in Dataflow Architectures

The application of software interactions for networking communications shares many commonalities with dataflow architectures (e.g. [64, 32]). Both approaches interconnect components (also referred to as "nodes") via links and port-like structures to form a complete system. However, key differences exist between the two approaches. The two approaches differ in the kinds of nodes and links they support, the kinds of topological changes they permit, and the means of execution they apply.

The nodes of a dataflow architecture are primitive and stateless while those of a software network are not. The nodes of a dataflow architecture are roughly equivalent to primitive instructions, while the nodes of software network may retain state and possess arbitrary computational power. The complexity of a system built with software interactions is defined not only by its network configuration, but also by the complexity of its nodes. The stateless nodes of a dataflow architectural are useful for simplifying the understanding and design of computational tasks, but they do so at the expense of efficiency. To preserve state information, developers must either employ greater numbers of nodes or re-formulate
their algorithms to use stateless elements; both of these tasks are usually difficult to do.

The links of a dataflow architecture are not reusable while software interactions are. The links of a dataflow architecture are abstract forms without concrete representation that are frequently identified with functional calls or message values. Software interactions, by contrast, are first-class entities that may be reused individually or as groups to recreate previous networks of communications. In addition, the links of a dataflow architecture are not normally produced to form multiple links between nodes while software interactions can and often do.

The topology of a dataflow architecture is static, while that formed from software interactions is dynamic. Software interactions can reorganize the configuration of a software network at run time. Thus, the operations of a model can often be made to resemble the true complexities of real-world systems. The conditional branching capabilities of dataflow architectures are inadequate to model systems that frequently undergo many topological changes. The conditional statements integrate, but do not separate, multiple configurations. Each configuration is defined as an integral part, not an operational state, of a system. Thus, as the number of configurations increases, the system becomes large and unwieldy.

Finally, the execution of a dataflow architecture is generally data-driven while a software network formed with software interactions is not. In a dataflow architecture, the availability of information pushes or pulls data through a system. In networks bound by software interactions, the interactions determine when and how data flows between computational components.

\[2\text{Because dataflow architectures are static, they are relatively easy to verify. In light of their dynamic connectivity, it may be that software interactions hamper the process of (software) verification. This question presents a topic for further research.}\]
Chapter 13

Conclusion

This chapter summarizes the work presented in this thesis and assesses its significance to the fields of visual simulation and computer science. The chapter begins with a brief overview of visual simulations and the challenges that hinder its development. Then, the chapter summarizes the thesis' objectives, goals, and contributions to the study of computer science. The chapter then discusses the impact of this work and the benefits it provides to further studies in visual simulation. Finally, the chapter identifies the drawbacks of this work and discusses potential approaches to enhance its design.

13.1 Visual Simulation

Visual simulations are computer programs that visualize the results of dynamic systems, systems that update their state in response to the advancement of simulation time. Many fields, such as computer graphics, medical imagery, and scientific visualization, use visual simulations to present the results of scientific study, to examine the behavior of complex systems, and to create the illusion of virtual interactivity.

Current programming tools and methods of visual simulation are capable of producing impressive results, such as films by Pixar Animation and the interactive rides of Disney Imagineering. Producing such results, however, typically involves much time and effort. Production times are long because current tools focus on helping developers produce prototypes. These tools provide few mechanisms for helping developers produce projects of larger size and complexity. In particular, current tools lack support for intermixing multiple animating methods, managing software complexity, and encouraging software reuse. In the absence of systematic approaches to each of these issues, developers typically devise their own methods, which may vary widely between applications and between individual developers. Individual
approaches cause problems of architectural mismatch and incompatible specifications. Moreover, it is seldom in the best interests of developers that they produce their own methods or toil for consistency across applications. By avoiding such duties, developers may focus their efforts in creating the content and specifying the behaviors of individual applications.

Current general programming tools and languages, such as object-oriented programming, are also not suitable for developing visual simulations efficiently. These tools are difficult to use for visual simulations because they lack sufficient abstract support for concurrency control and management of simulation time.

13.2 Summary of the Thesis

13.2.1 Objective

This thesis contends that visual simulations are easier to devise, manage, and reuse when using software interactions, first-class mechanisms for regulating the communications among software components. In lieu of function calls and conventional mechanisms for sharing information, the software interactions decide when and how the software components of visual simulations communicate. With software interactions, the development of visual simulation involves the application of simulation modeling to the building of software networks, an arrangement of software interactions interconnecting software components. As software interactions perform over time, software networks undergo changes affecting their topologies and periods of activation. This design improves upon the existing methods of visual simulation in that it enables developers to devise and manage separately structures for computation and coordination. The former determines what visual simulations compute while the latter determines how and when the visual simulations apply their computations to advance state.

13.2.2 Solution

This thesis validates its contentions with the presentation of interaction-based simulation (IBS), a novel approach to visual simulation that applies software interactions to facilitate the development, management, and reuse of visual simulations. IBS eases the development of visual simulations by supporting software composition, the building of software systems with plug-compatible components. The organization of components and interactions determine how visual simulations compute and animate state. IBS simplifies the management of visual simulations by providing multiple ways of addressing complexity, such as those involving constraints and multiple methods of simulation modeling. IBS enables the reuse of visual
simulations by encouraging a separation between the parts of a program that compute and those that coordinate. Each part develops independently to operate easily with the other.

**Interaction-Based Simulation**

In this thesis, the presentation of IBS consists of a description of its parts and its application to an illustrative example (chapters 3-8), an evaluation of empirical results (chapter 9-10), an assessment of utility (chapter 11), and a discussion of background material and related work (chapters 2,12).

The individual parts of IBS consist of *interaction-based programming* (IBP), *interaction constraints*, *multi-modeling methods*, a general specification, and *reuse guidelines*. As illustrated in Figure 13.1, the five parts assemble into three layers, with IBP at the bottom and the general specification at the top. IBP is a programming approach that constructs *scripts* and *meta-scripts* to govern the execution of software interactions. Scripts assemble software networks while meta-scripts coordinate the operations of scripts. Interaction constraints establish dependencies between software interactions or among software components. The dependencies relate the software interactions by their execution or by their dealings with software components. They relate software components by binding the interface of the components with numerical operators. Multi-modeling methods intermix with the scripting of software interactions. Each method applies an alternative approach to composing software networks from software interactions. The general specification establishes a general approach to describe the content and behavior of a visual simulation. Applying the first three parts of IBS, the general specification partitions visual simulation into the definition of three graphs. Reuse guidelines encourage the development of reusable components. The guidelines develop software components to support software interactions and the IBP programming environment.

![Figure 13.1: The three layers of Interaction-Based Simulation](image)

The evaluation of empirical results involves the comparison of IBS, as embodied by
CHAPTER 13. CONCLUSION

the RASP toolkit, with popular tools for building visual simulations. The RASP toolkit is a stylized C++ programming tool that applies software interactions to the interconnection of components. The empirical results consist of several implementations of several scenes. Each scene either encourages the use of an alternative approach to animating state, or presents the identifying characteristics of a domain that applies animated visuals. The comparison presents the capabilities of IBS to produce scenes of wide variety and of varying domains, and the advantages of IBS in managing software complexity and supporting software reuse.

The assessment of utility focuses on IBS’s ability to ease the development of visual simulations. Using analytic arguments and empirical evidence, it discusses the benefits of dynamic topology, multi-modeling, and the reuse guidelines. With dynamic topology, visual simulations are free to assume any configuration, not only those set by its implementation. With multi-modeling, visual simulations develop incrementally. With the reuse guidelines, the components of visual simulation are more useful and operate well in multiple contexts.

The discussion of background material and related work provides a context against which to evaluate IBS and its advantages. The discussion on background material conveys the aims of visual simulation and presents various techniques for building them. The discussion on related work differentiates software interactions from the mechanisms of previous approaches. The discussion shows how software interactions are unique, especially for visual simulation, and how software networks, which arise with the use of software interactions, are unlike dataflow architectures.

13.2.3 Contributions

This thesis makes three significant contributions to the study of computer science and visual simulation. First, it introduces IBS, a novel approach to visual simulation, and demonstrates its usefulness. With IBS, the development of visual simulations involves the building of two activities, activities for production and activities for coordination. Each of the activities addresses a separate concern in the development of visual simulations, that may be effectively produced and managed with alternative means. The activities for production, which establish state and identify communications, permit visual simulations to be developed from a collection of parts, while the activities for coordination, which produce and manage the execution of software interactions, enable visual simulations to develop as a progression of software networks with changing topologies.

Second, the thesis presents new approaches to manage the complexity of visual simulations and presents the usefulness of each approach in simplifying the development of animated scenes. Leveraging the design benefits of IBS, visual simulations become easier to
manage with multi-modeling methods, interaction constraints, and a general specification. Multi-modeling methods, which integrate simulation techniques with computer animation, permit visual simulations to intermix animating methods and to produce animated scenes iteratively. Developers select and apply the methods that best model the needs of individual scenes. Interaction constraints, which apply dependencies between components or software interactions, simplify the monitoring and managing of communications in visual simulations, especially those that grow in size and complexity. As communications flow between components, the interaction constraints support the application of numerical operators and logical functions to the values being communicated, or to the order in which communications occur. The general specification partitions visual simulations into three distinct graphs, and eases the development of visual simulations by parts. Each graph develops from the use of software composition and encourages the use of a general file format for visual simulation that is logical, extensible, and device-independent.

Third, the thesis introduces design guidelines for improving the reuse of components that comprise visual simulations, and demonstrates the guidelines' effectiveness using empirical results. The design guidelines, which consist of three sets of properties, prepare components to operate consistently as software interactions control their communications. The guidelines improve the ability of components to operate in varying environments, which may change with the advancement of simulation time or with the introduction of new applications. Components that are not bound to support a specific environment or a specific type of visual simulation become easier to reuse.

13.3 Impact

This thesis and its use of software interactions suggests an assortment of research projects that apply explicit controls to the interconnecting of software components. Here follows a short discussion describing several endeavors in the fields of visual simulation and software development that may benefit from the work of this thesis.

13.3.1 Visual Simulation

For visual simulation, this thesis introduces three benefits: a means to extend VRML to support animation effectively, a means to introduce a visual interface to visual simulation, and a means to experiment with new forms of animation.
Extension to VRML

The general specification of chapter 7 complements and improves the design of VRML in animating virtual scenes. VRML, in its current form, supports a non-systematic approach to animation. It applies routes and scripting nodes to interconnect components and perform numerical calculations. Unlike the scene graph of VRML, which is useful in rendering and modeling geometric shapes, the routes and scripting nodes of VRML are insufficient to support a systematic approach to animation. Routes are applied haphazardly without any means or rules to organize their use or to vary their application over time. Scripting nodes simply encapsulate the calls to a scripting language without providing any mechanisms for managing parallelism or managing the advancement of time. The general specification of IBS, in contrast, supports a systematic approach, with both an action graph and a time graph. The action graph replaces the routes of VRML with a hierarchical structure to plan the interconnection of fields and nodes, and the time graph introduces a temporal ordering to the interconnection of nodes and the development of time-varying systems.

Visual Programming

IBS supports the application of visual programming to visual simulation. With visual programming, developers apply an interactive interface to the building of software applications. With IBS, the visual interface for visual simulation could involve the building of the three graphs of the general specification, or the construction of software networks, such as those appearing throughout this thesis (i.e., Figure 4.3). A visual representation of the three graphs would be helpful in developing the basic structure of a visual simulation. It would provide developers with an interactive means of assembling and recognizing an ordering of nodes that develops a particular shape, behavior, or temporal specification. A visual representation of a software network would also likely be helpful in developing the basic architecture of a visual simulation. It would provide developers with an interactive approach to assemble and coordinate the communications between a collection of software components.

Experimental Animation

IBS encourages the development of new approaches in computer animation. Areas of potential research include the development of an adaptive animation or the derivation of a qualitative animation. An adaptive animation adjusts its precision and quality according to run-time conditions, such as the changing distance between a camera and the animating elements of a scene. As the distance grows, the animation applies fewer components or fewer
software interactions to animate the behaviors of the scene. Similar to a geometric model, which may contain multiple levels of detail [128], an adaptive animation may store multiple graphs (of the general specification), with each graph representing a separate approach to advancing time and state.

A qualitative animation characterizes the general behavior or basic qualities of an animated scene. The general behavior or basic qualities of a scene could be obtained by repeatedly varying the use of components and software interactions or by refashioning the temporal ordering of scripts. Applied with stochastic techniques and statistical methods, the varying use of components, software interactions, and scripts is helpful in analyzing the variations of a scene and in quantifying the results of uncertainty or the occurrence of variable events. For example, a qualitative animation may repeatedly test and analyze the conditions of a software network that control the movements of a geometric figure. From its analysis, the animation may present statistical results, such as the spatial range of the figure's movements, or signal erroneous configurations, such as those that cause the figure to fall.

13.3.2 Software Development

This thesis introduces a new approach to software development. With IBS, software programs arise from the building of scripts and meta-scripts that control the execution of software interactions. The application of scripts and meta-scripts to software development goes beyond that of simply producing visual simulations. They also offer great potential in developing new approaches to programming soft real-time applications, coping with distributed computations, and constructing configurable systems.

In programming soft real-time applications, the meta-scripts could vary the application of scripts and software interactions to real-time operating constraints. If one software network is too slow, a faster software network could be applied as a replacement. The meta-scripts may also apply multiple scripts in parallel to race several software networks of varying complexity. The network of greatest complexity that finishes under an acceptable time is applied and communicated first. In coping with distributed computations, the scripts or meta-scripts could apply rules to partition the computations of a program over a physical network. The rules would decide when and how the elements of a network share information while the software interactions communicate information over the network for the components to send and receive.

In producing a configurable system, the meta-scripts could apply rules to construct a script dynamically. The script would decide the configuration of a software network at
run-time. A configurable system is important for developing programs that may need to change over time without human intervention. For instance, the system for a satellite involved in deep-space exploration may wish to re-configure itself as it accumulates damages or encounters failures in its parts. A meta-script could configure a script that replaces the primary parts of a system with viable substitutes. A configurable system is beneficial in that it incorporates only those parts that operate and only those that are necessary. In addition, a configurable system could conceivably address a wider range of problems than could a single program in that the configurable system could adapt itself in response to run-time conditions.

13.4 Future Investigation

The current design of IBS invites many areas of future investigation. IBS is not without drawbacks, and solutions are needed to improve its design so that it truly models, manages, and reuses a wide variety of scenes. The remainder of this chapter discusses some of the current drawbacks of IBS, a proposal for possible enhancements, and a commentary on user studies.

13.4.1 Drawbacks

IBS is an effective approach for producing a wide variety of visual simulations. However, it is not a panacea for resolving the challenges of all types of visual simulations. IBS is unable to create visual simulations that introduce run-time actions, to encourage consistency amongst the design of its meta-scripts, and to attest clearly to the reusability of its development.

Run-Time Actions

The current design of IBS does not readily support adequate means to spawn run-time actions, actions occurring in response to run-time conditions. Run-time actions are essential for blending simulation technologies with animation techniques. In process-oriented simulations [169], actions occur in response to special events, referred to as transactions. Each transaction earmarks the arrival of certain states or the recent availability of resources. In animation, explosions are good examples of run-time actions. They are unpredictable and may occur often. The modeling of several explosions may be handled as a whole with a central set of components, or as several instances with a collection of individual components. The former approach uses components of greater complexity than that of the latter. The
latter approach, as favored by the simulation community, is best for developing larger sys-
tems of greater complexity. If individual explosions should behave differently over time and
become involved in different experiences, it is best that each be managed as an individual,
not as one part of a collective unit. The collective unit grows complex quickly if it attempts
to accommodate the wide variability associated with each explosion. Unfortunately, to pro-
duce and schedule each explosion as an individual action is tedious and inefficient with IBS.
The current design of IBS requires that each explosion be predefined and readied to occur.
It would be far better if IBS could specify one action that spawns each time an explosion
occurs.

IBS is unable to spawn run-time actions because its time graph is not designed to
handle actions that arise haphazardly with unknown repetition. To develop properly, a time
graph requires a priori knowledge of all actions, including their being and representation.
Unfortunately, run-time actions are linked to the execution of a simulation and are unknown
until they are required. A possible solution to this drawback could involve the introduction
of spawning nodes, which could generate new actions in response to the emergence of a given
state or behavior. For instance, whenever the value of a virtual port reaches zero, as would
occur if the port computes the distance between colliding objects, a spawning node could
invoke a secondary action that ignites a spark at the contact point between the two objects.

Consistency among Meta-Scripts

The current design of IBS opens up the possibilities of a wide range of meta-scripts. With too
much diversity and few rules to decide how scripts and meta-scripts develop, the complexity
of visual simulations may become difficult to manage and apply consistently. For example,
the specification of a time graph, as applied by the general specification of IBS, does not
apply a simple and consistent approach to the use of conditionals, which associate the timing
of one script to the activity of another. Currently, all scripts must specify, prior to their
use, what information meta-scripts may apply in developing a conditional. For instance, if
a meta-script wishes to apply a conditional to the positioning of an object, such that the
object should cause a light to shine, the meta-script must expect any script that moves the
object to compute the relations of the conditional and then to provide a boolean value (via
a StateEvent).

This approach to developing conditionals is awkward in that it requires a script to
accommodate the needs of a single context, not of a wide range of scenarios. In addition, it
is not conducive to supporting complex conditionals, which associate the timing of one script
to the activities of several. It would be simpler for design and reuse if meta-scripts did not
cause scripts to support any conditionals. One way to achieve this goal could involve the use of wrapper scripts, special scripts that envelop one or more scripts with logical expressions, which meta-scripts may apply in evaluating a conditional. The logical expression would access the ports or interactions of the enveloped scripts by their names, types, or by the kinds of values each communicates. To apply a conditional to the positioning of a moving object, for instance, the wrapper script could envelop the script that moves the object with a logical expression that accesses the port causing the object to move.

**Reusability**

With the current design of IBS, three problems arise concerning software reuse. First, it is unclear if all of the properties of the reuse guidelines are satisfiable and, moreover, practical. The partitioning of multi-parameter operations into an elementary set of unary operations, for instance, may inadvertently add clutter to the definition of a component and introduce confusion to the usage of the original operations. To avoid such difficulties, it may be necessary to introduce special rules that either justify the usage of specific properties or clarify the usage of alternatives. Second, troubles shall undoubtedly arise when components that violate the guidelines are applied for use with IBS. In its current form, IBS lacks guidelines for handling these components and accommodating their use. A solution to this problem may either envelop such components with wrappers or confine these components to specific contexts. The wrappers would attempt to bring the components inline by offering services and mechanisms, such as ports and totality measures. In confining the components to specific contexts, such as those that update time with a constant timestep, specific properties need not be observed. Third, it is not clearly evident whether or not the scripts of IBS are truly reusable. Scripts are mostly self-contained and separate from meta-scripts. Hence, as discussed in Lee [87], the scripts may conceivably be easily reused and applied readily to different applications. Further testing is needed to quantify this assumption and to determine if this undertaking is pragmatic.

### 13.4.2 Enhancements

The utility of IBS should improve with the introduction of practical enhancements. Four possibilities include the introduction of extensions for interactions constraints, the development of better abstractions for multi-modeling, the production of debugging tools, and the specification of a language for interaction-based simulation.
CHAPTER 13. CONCLUSION

213

Extensions to Interaction Constraints

The interaction constraints of IBS relate ports (and software interactions) with one-way constraints, which update only upon instruction. A natural extension to interaction constraints involves the use of two-way constraints and the use of constraints, either one-way or two-way, that update automatically. Both extensions could simplify the development of scripts that create software networks of moderate complexity. Two-way constraints minimize the application of many one-way constraints, in that two-way constraints form mutually dependent relations amongst ports. If any port of a two-way constraint changes state, all remaining ports update accordingly. For instance, if the sum of any two ports is constant, a change in one causes a change in the other. Automatic constraints diminish the use of interactions in maintaining fixed relations. Rather than waiting for an interaction to request a value, the constraints update immediately and communicate their results accordingly. For instance, in fixing a camera to focus on an object, a two-way constraint updates the orientation of the camera immediately after the object moves. The script that sets the constraint need not apply an interaction to ensure that the camera is updated appropriately.

Abstractions for Multi-Modeling

The multi-modeling methods of IBS apply abstractions to simplify the intermixing of animating methods with the techniques of computer simulation. The current design of these abstractions may or may not accurately reflect the design of the original algorithms and data structures that they imitate. For example, the finite state automaton of Figure 6.5 consists of one abstraction that might be of greater use if it were produced from several parts. Each of the parts could be developed as a separate state with its own set of interactions. The states could then be grouped with conditional or temporal values to reproduce the workings of a deterministic or non-deterministic finite state automaton.

Debugging Tools

As with any new programming approach, there is always an expectation that developers will produce errors in their code. Hence, it is important to offer debugging utilities that are consistent, effective, and simple to apply with IBS. The current version of RASP provides basic abstractions for debugging, such as TEXTEVENT and INPRINTINFO. The former is a software interaction that streams user-defined text to a display or file, and the latter is a standard port that streams the state values of a component to a display. TEXTEVENT is helpful in tracing the execution of scripts while INPRINTINFO is helpful in presenting the run-
time state of components. RASP also provides basic support for detecting run-time errors. For instance, RASP signals a run-time error if multiple scripts write simultaneously to the same port. Simultaneous writes to a signal port are impossible in time-varying systems since state variables may update only once per time step. Future extensions to RASP may include further debugging and robustness features, such as exception handling, that may prove useful in easing the difficulties of applying IBS to visual simulations. Exception handling could be applied to unwind the execution of software interactions and to allow corrective actions to take place at the problem source.

**Interaction-Based Language**

As can be seen from the examples of chapters 9 and 10, the syntax of RASP is unconventional and involves a greater learner curve. The development of a new language, specifically designed for interaction-based simulation, may prove useful in simplifying the development of scripts and encouraging further improvement to the design of IBS. A scripting language could also be introduced to hide some of the details of IBS. Similar to how MEL, the scripting language of MAYA, hides the details of building a dependency graph, a scripting language for IBS could hide the details of forming a software network.

**13.4.3 User Studies**

User studies, performed with great care, would be useful in producing anecdotal evidence to the utility of IBS. They generate empirical results, create a foundation to derive logical arguments, and signal a list of drawbacks. In evaluating IBS, user studies can help assess the quality of its design, its features, and its performance.

User studies can assess the quality of IBS’s design by examining its ease in developing visual simulations. The studies can determine if the syntax of IBS, as embodied by RASP, is difficult to apply. The use of interactions and ports is unconventional, and the use of C++ to produce software networks is unusual. A difficult syntax undoubtedly distracts developers from focusing their efforts on their other tasks. The user studies could also identify certain types of scenes or actions that are consistently difficult to produce. As mentioned previously, scenes that spawn run-time actions are not easily produced with the current design of IBS.

User studies can also evaluate IBS’ features. For instance, multi-modeling assembles a software component by integrating a script with proxy ports. Further testing will help evaluate the utility of this action and its usefulness for producing a multi-model scene. It may be quicker and possibly simpler, for example, to reuse a script by cutting and pasting...
its code, rather than by writing code to interface a script with proxy ports. Other features, such as the reuse guidelines, may be assessed with experiments geared to determine the effectiveness of the reuse guidelines in building reusable components.

User studies can assess the performance of IBS by developing tests of speed and identifying critical bottlenecks. In visual simulation, speed is often critical to convey a sense of realism and to enable interactive effects. The identification of bottlenecks is important to guide the development of future enhancements, which may conceivably include the introduction of RASP-lite, a lightweight toolkit operating with fewer features but superior performance.
Bibliography


Appendix A

Source Code Supplement

The source code appearing in this appendix supplements the code presented in chapters 9 and 10. This code implements various functions and defines numerous variables for completing the implementation of individual scenes. The line numbers, appearing on the left, identify the placement of the code into a scene. Code beginning at 1 appears as a preamble, while code beginning with larger values appears as an ending.

A.1 Animating Methods

A.1.1 Key-Frame Animation with Cory

The code appearing in Figures A.1 and A.2 completes the coding of the example of section 9.2.1. The code of Figure A.1 defines ball_cue, a cue animating the position of a ball. The code is similar to that appearing in Figure 9.2. ball_cue accesses three evaluators to individually set the three dimensions of the ball’s position. The code of Figure A.2 completes the partial listing of Figure 9.4 in presenting seven evaluators and three spline-based datasets. The code of Figure 9.2 accesses the seven evaluators to interpolate and scale the datasets so as the animate the position and orientation of a camera and a ball. Each of the datasets consists of three values with three times. Collectively, the three datasets represent three keyframes.¹

¹At the time of Cory’s implementation, object-oriented techniques were not as prevalent as they are today. A modern implementation of Cory could apply higher order abstractions, such as points and splines, to simplify the coding of Figures A.1 and A.2, such that each dataset could control multiple dimensions, not just those of x, y, or z.
animate the position of the ball

(55)  ball.cue := cue {
    perform when cue starts

    (56)    start_action = (;
    (57)        "ball.x.t.offset = (ball.cue start?)",
    (58)        "t.scale = (ball.cue duration?)",
    (59)        "ball.y.t.offset = (ball.cue start?)",
    (60)        "t.scale = (ball.cue duration?)",
    (61)        "ball.z.t.offset = (ball.cue start?)",
    (62)        "t.scale = (ball.cue duration?)";

    perform as time advances

    (63)    tick_action = (;
    (64)        "Ball position = (ball.x.value @ (ball.cue time?),
    (65)            " ball.y.value @ (ball.cue time?),
    (66)            " ball.z.value @ (ball.cue time?)");

};

Figure A.1: An implementation in CORY

A.1.2 Procedural Animation with Mam/VRS

The code of Figure A.3 completes the coding of the example of section 9.3.1. The code presents the class definition for the XtRobot, whose constructor and method are presented in Figures 9.9 and 9.10. The class definition consists of nine private variables, of which two define pucks (line 3), five define three behaviors (line 4-8), and two define local variables for fsaUpdate, the class's single method (line 9-10).

A.1.3 Procedural Animation with Fran

The code in Figure A.4 completes the coding of the example of section 9.3.2. The code defines the shape, appearance, and location of two pucks and one robot (lines 1-5). The operator **/ integrates geometrical information with physical attributes, such as size and color.

A.1.4 Particle System Animation with Asas

The pseudo-code in Figure A.5 completes the coding of the example of section 9.4.1. The code begins with the definition of a moving pole and a variable storing the pole's radius (lines 1-4). The code of Figure 9.16 applies the pole's radius to determine if a particle is too close to the proximity of the pole. The code follows with a definition of offset, a function for updating the position of a particle near the pole (lines 5-10). offset computes a new
```plaintext
create evaluators
(68) cam.x := eval {
(69)    data_name = "dataX"
(70)    v.scale = 1.0;
(71) }
(72) cam.y := eval {
(73)    data_name = "dataY"
(74)    v.scale = 1.0;
(75) }
(76) cam.z := eval {
(77)    data_name = "dataZ"
(78)    v.scale = 1.0;
(79) }
(80) look.y := eval {
(81)    data_name = "dataY2"
(82)    v.scale = 1.0;
(83) }
(84) ball.x := eval {
(85)    data_name = "dataX"
(86)    v.scale = 1.0;
(87) }
(88) ball.y := eval {
(89)    data_name = "dataY"
(90)    v.scale = 1.0;
(91) }
(92) ball.z := eval {
(93)    data_name = "dataZ"
(94)    v.scale = 1.0;
(95) }
create spline-keyframes
(96) dataX := spline_data {
(97)    init.der = 0.0
(98)    final.der = 0.0
(99)    time = 0.0 value = 32.0
(100)   time = 0.5 value = 32.0
(101)   time = 1.0 value = 16.0
(102) }
(103) dataY := spline_data {
(104)   init.der = 0.0
(105)   final.der = 0.0
(106)   time = 0.0 value = 75.0
(107)   time = 0.5 value = 75.0
(108)   time = 1.0 value = 22.0
(109) }
(110) dataY2 := spline_data {
(111)   init.der = 0.0
(112)   final.der = 0.0
(113)   time = 0.0 value = 85.0
(114)   time = 0.5 value = 85.0
(115)   time = 1.0 value = 32.0
(116) }
(117) dataZ := spline_data {
(118)   init.der = 0.0
(119)   final.der = 0.0
(120)   time = 0.0 value = 18.0
(121)   time = 0.5 value = -18.0
(122)   time = 1.0 value = 50.0
(123) }
(124) }
```

Figure A.2: An implementation in CORY

position for veering the particle away from the pole's center.

A.1.5 Particle System Animation with RASP

The code in Figure A.6 completes the coding of the example of section 9.4.3. The code begins with the definitions of two random number generators for producing three dimensional values (lines 1-2). Applied by the code of Figure 9.18, the generators serve either to jitter the movements of a particle so as to avoid a moving pole or to start a particle at a random location. The remainder of the code establishes seven port groups, three for unidirectional write and four for unidirectional read (lines 3-9). The port groups consolidate the ports of individual particles and of motion controllers such that individual software interactions may access all the ports as a single group. The addPort method of a port group binds a port to a
class XtRobot : public MXtAppFrame
{
private:
(1) puck = importX "puck.x"
(2) puck1 = translate3 (vector3XYZ (-2) 0 0) *** uscale3 0.3 *** withColorG red puck
(3) puck2 = translate3 (vector3XYZ 2 0 0) *** uscale3 0.25 *** withColorG blue puck
(4) robotX = importX "robot.x"
(5) robot = translate3 (vector3XYZ 0 0 0) *** uscale3 0.2 *** withColorG green robotX

public:
(12) XtRobot(); // constructor
(13) void fsaUpdate(); // method
};

Figure A.3: An implementation in MAM/VRS

Figure A.4: An implementation in FRAN

group (lines 13-20), while the outAllPorts method returns a port representing all the values of the group (i.e., line 23 of Figure 9.18). The motion controllers, of type LawsOfMotion, animate the movements of the particles with physical laws (lines 11-12).

A.2 Domain Applications

A.2.1 Algorithmic Animation with Polka

The code of Figure A.7 provides a complete implementation of the coding in Figure 10.3 of section 10.2.1. The code presents the class interfaces for the Rects, a subclass of View, and MyAnimator, a subclass of Animator. Rects provides four functions to operate on twenty columns. The first two functions initialize the heights of the columns and the latter two functions animate two of the columns to exchange positions. myAnimator interprets the events of an animation and invokes the appropriate actions of Rects.
define pole
(1) (define poleRadius4 ... )
(2) (defop pole
(3) (actor (local: (....))
(4) (... motion instruction .. )))
compute
(5) (defop offset
(6) (param: distance position deltaV)
(7) (local: (fact (div poleRadius4 dist))
(8) (diff (times (normalize (dif position polePosition) fact)))
(9) (offset (plus (plus (diff position) (vector <randX> <randY> <randZ>)))))
(10) (plus offset deltaV))

Figure A.5: An implementation in Asas

random number generators
(1) RandomPt3 *rand = new RandomPt3( Point3(-1,0,0), Point3(1,0,0) );
(2) RandomPt3 *rand2 = new RandomPt3( Point3(-5,2,-130), Point3(5,3.5,-105) );
assemble ports into groups
(3) OPortGroup *groupO = new OPortGroup;
(4) IPortGroup *groupI = new IPortGroup;
(5) OPortGroup *motions = new OPortGroup;
(6) OPortGroup *motionP = new OPortGroup;
(7) OPortGroup *motionV = new OPortGroup;
(8) IPortGroup *physicP = new IPortGroup;
(9) IPortGroup *physicV = new IPortGroup;
load ports of particles into groups
(10) for(int i=0; i<100; i++) {
(11) MotionLaw *motion = new LawsOfMotion;
(12) Physics *pInfo = (Physics*) particles[i]->getAttribute( Physics::getClassName() );
(13) groupO->addPort( particles[i]->outPosition() );
(14) groupI->addPort( particles[i]->inPosition() );
(15) motions->addPort( motion->inTime() );
(16) motionP->addPort( motion->outNewPos() );
(17) motionV->addPort( motion->outNewVel() );
(18) physicP->addPort( pInfo->inPosition() );
(19) physicV->addPort( pInfo->inVelocity() );
(20) }

Figure A.6: An implementation in Rasp

A.2.2 Algorithmic Animation with Obliq-3D

The two procedures of Figure A.8 complete the coding of the example of section 10.2.2. 
class MyAnimator: public Animator {
public:
MyAnimator() {}

handle events
int Controller() {
  if (!strcmp(AlgoEvtName,"Init"))
    r.Init();
  if (!strcmp(AlgoEvtName,"Input"))
    r.Input(AnimInts[0],AnimInts[1]);
  if (!strcmp(AlgoEvtName,"Show"))
    r.Ready(AnimInts[0]);
  if (!strcmp(AlgoEvtName,"Swap"))
    r.Swap(AnimInts[0],AnimInts[1]);
  return(1);
}
private:
Rects r;
};

class Rects: public View {
public:
Rects() { max=0; }
initialize view
int Init() {
  Create("SelectionSort");
  return (0);
}
  save values in the array
  int Input( int index ,int val ) {
    values[index] = val;
    if (val > max) max=val;
    return(1);
  }
  function declaration
  int Ready(int);
  int Swap(int,int);
private:
  int max, values[20];
  Rectangle *blocks[20];
};

Figure A.7: An implementation in POLKA

in associating themselves with a particular index of the sorted array.

A.2.3 Scientific Visualization with RASP

The code appearing in Figure A.9 completes the coding of the example of section 10.3.3. The code consists of two procedures, changeColor (lines 1-11) and renderModel (lines 29-32). changeColor animates the color of a model by interpolating the model's material attribute. An interpolator produces a sequence of values from zero to one, for the interaction clrEvt to communicate to the material's color port. To update only the red component of the color, clrEvt performs a partial write when communicating the values to the port. The interaction maintains the original "green" and "blue" elements that the port expected (lines 6-7). The events durEvt and tEvt, sequenced by the activity act, provide the interpolator with temporal information over time (lines 4-5). renderModel decides when to render a model to the viewport (lines 29-32). The code of Figure 10.18 applies renderModel to animate the visibility of the sphere and its bounding box over time.
APPENDIX A. SOURCE CODE SUPPLEMENT

obtain index of a value

(46) let getlndex = proc( C, id )
(47) var index = -1;
(48) for j=0 to #(C)-1 do
(49) if C[j].id is id then
(50) index := j;
(51) end;
(52) end;
(53) index;
(54) end;

swaps two indices

(55) let swaplndices = proc( C, x, y )
(56) var temp = 0;
(57) temp := C[x].id;
(58) C[x].id := C[y].id;
(59) C[y].id := temp;
(60) end;

Figure A.8: An implementation in OBLIQ-3D

Activity* changeColor( Model *outline )
{
    access color attribute
    (36) Material *mat = (Material*) outline—»getInformer( Material::getClassTypeId() );
    (37) ColorBase *clr = mat—»getDiffuse();
    interpolator and events to update color
    (38) EvolveDbl *ev = new EvolveDbl( 0, 1 );
    (39) DurationEvent *durEvt = new DurationEvent( ev—»inDurationVal() );
    (40) TimeEvent *tEvt = new TimeEvent( ev—»inDurationVal() );
    partial write color attribute
    (41) ClrConnect *filter = new ClrConnect( ColorBase::COMP1 );
    (42) Event *clrEvt = new Event( ev—»outCurVal(), clr—»inDiffuse(), filter );
    organize software interactions
    (43) Activity *act = new Activity();
    (44) act—»addInitEvent( durEvt );
    (45) act—»addActEvent( tEvt, clrEvt );
    (46) return act;
}

communicate model ready for rendering

Activity* renderModel( RaspSetting *world, Model *model )
{
    (47) Event *solEvt = new Event( model—»outThis(), world—»inRender() );
    (48) Activity *act = new Activity();
    (49) act—»addActEvent( solEvt );
    (50) return act;
}

Figure A.9: An implementation in RASP
Appendix B

EBNF Supplement

The following EBNF specification supports the general specification of chapter 7.

B.1 General

\[
\begin{align*}
\text{raspScene} & ::= \text{statements} ; \\
\text{statements} & ::= \text{statement} | \text{statement statements} | \text{empty} ; \\
\text{statement} & ::= \text{modelStatement} | \text{actionStatement} | \text{timeStatement} ; \\
\text{empty} & ::= ; \\
\text{Id} & ::= \text{any number of characters} \\
\text{constant} & ::= \text{any numerical value}
\end{align*}
\]

B.2 Model Graph

B.2.1 Nodes

\[
\begin{align*}
\text{modelStatement} & ::= \text{graphNode} | \text{DEF modelName graphNode} | \text{USE modelName} ; \\
\text{graphNode} & ::= \text{nodeType} \{ \text{nodeBody} \} ; \\
\text{nodeType} & ::= \text{Id} ; \\
\text{nodeBody} & ::= \text{nodeBodyElement} | \text{nodeBodyElement nodeBody} | \text{modelStatement} | \text{empty} ; \\
\text{nodeBodyElement} & ::= \text{dataType fieldId fieldValue} | \text{portType portName dataType} ; \\
\text{dataType} & ::= \text{RS_BOOL} | \text{RS_INT} | \text{RS_DOUBLE} | \text{RS_POINT3} | \text{RS_STRING} | \text{RS_VECTOR} | \ldots ; \\
\text{fieldId} & ::= \text{Id} ; \\
\text{portName} & ::= \text{Id} ; \\
\text{modelName} & ::= \text{Id} ;
\end{align*}
\]
B.2.2 Fields and Ports

```
portType ::= OPORT | IPORT ;
fieldValue ::= boolValue | intValue | doubleValue | point3Value | stringValue | ... ;
boolValue ::= TRUE | FALSE ;
intValue ::= (+)?((0[0-9]+)|0[xX][0-9a-fA-F]+))
doubleValue ::= ([+/-]?(((0-9)*0-9)+))([eE][+]?0-9)+)
point3Value ::= doubleValue doubleValue doubleValue ;
stringValue ::= ".*"
```

B.3 Action Graph

```
actionStatement ::= actionNode | DEF actionName actionNode | USE actionName ;
actionNode ::= actionType { actionBody }
actionType ::= Id ;
actionBody ::= LOCALS { localBody } | EVENTS { eventsBody } | VIRTUALS { virtualBody } | ACTIVITIES { actBody } ;
actionName ::= Id ;
```

B.3.1 Locals

```
localBody ::= modelStatement ;
```

B.3.2 Events

```
eventsBody ::= eventsBodyElement | eventsBodyElement eventsBody | empty ;
eventsBodyElement ::= interactionNode | DEF interNodeName interactionNode | USE interNodeName ;
interactionNode ::= interType { interBody } ;
interType ::= Id ;
interBody ::= interBodyElement | interBodyElement interBody | empty ;
interBodyElement ::= fieldId interFieldValue ;
interFieldValue ::= portRef | modelName | { eventsBodyElement } ;
interFieldName ::= Id ;
portRef ::= [ modelName portName ] ;
interNodeName ::= Id ;
```
B.3.3 Virtuals

\[
\begin{align*}
\text{virtualBody} &::= \text{virtualNode} \mid \text{virtualNode virtualBody} \mid \text{empty} \\
\text{virtualNode} &::= \text{virName} = \text{virtualDef} \\
\text{virName} &::= \text{Id} \\
\text{virtualDef} &::= \text{virArg} \mid \text{virtualDef operator virtualDef} \mid \text{vpFunction (virList)} \\
\text{virArg} &::= \text{portRef} \mid \text{virName} \mid \text{constant} \\
\text{vpFunction} &::= \text{Id} \\
\text{virList} &::= \text{virArg} \mid \text{virArg, virList} \mid \text{empty}
\end{align*}
\]

B.3.4 Activities

\[
\begin{align*}
\text{actBody} &::= \text{SET assocName orderType \{} \text{threads} \} \\
\text{assocName} &::= \text{Id} \\
\text{orderType} &::= \text{Id} \\
\text{threads} &::= \text{thread} \mid \text{thread threads} \mid \text{empty} \\
\text{thread} &::= \text{Thread constant interactions} \\
\text{interactions} &::= \text{eventsBodyElement} \mid \text{eventsBodyElement, interactions}
\end{align*}
\]

B.4 Time Graph

\[
\begin{align*}
\text{timeStatement} &::= \text{timeNode} \mid \text{DEF timeName timeNode} \mid \text{USE timeName} \\
\text{timeNode} &::= \text{timeType \{} \text{timeBody} \} \\
\text{timeType} &::= \text{Id} \\
\text{timeBody} &::= \text{timeFields timeBodyElements} \\
\text{timeFields} &::= \text{timeField} \mid \text{timeField timeFields} \mid \text{empty} \\
\text{timeField} &::= \text{dataType fieldId fieldValue} \\
\text{timeBodyElements} &::= \text{timeBodyElement} \mid \text{timeBodyElement, timeBodyElements} \mid \text{empty} \\
\text{timeBodyElement} &::= \text{timeStatement} \mid \text{USE actionName} \\
\text{timeName} &::= \text{Id}
\end{align*}
\]
Glossary

**action graph** - a hierarchy of nodes that sequences the execution of software interactions.

**dynamic topology** - configurational changes to a software network over time.

**first-class operation** - a method (or function) of a software component possessing identity and state.

**general specification** - a partitioning of a visual simulation into three graphs: model, action, and time.

**interaction-based programming (IBP)** - a programming approach applying software interactions to mediate the communications between software components.

**interaction-based simulation (IBS)** - a developmental approach for producing visual simulation with software interactions.

**interaction constraint** - a first-class association that forms either a virtual event or a virtual port.

**model graph** - a hierarchy of nodes that defines the geometry and material characteristics of shapes.

**meta-script** - any abstraction that coordinates the execution of scripts (or other meta-scripts).

**script** - any abstraction that coordinate the execution of software interactions.

**software composition** - the building of code with abstractions that generalize objects, agents, and functions.

**software interaction** - a first-class mechanism for mediating the communications between (the operations of) software components.

**software network** - a collection of components bound together with software interactions.
Glossary

**time graph** - a hierarchy of nodes that organizes the execution of scripts over time.

**reuse guidelines** - a set of properties for improving the reuse of software components that operate in an interaction-based environment.

**unidirectional port** - a first-class operation of a component that either provides or receives information.

**virtual port** - a port, unassociated with any component, that relates the ports of components with mathematical operator and numerical functions.

**virtual interaction** - an interaction that relates the execution of several software interactions.

**visual simulation** - any time-varying program that visualizes its state.
Index

animating methods, 4
  key-frame animation, 124
particle system animation, 143
procedural animation, 134
animation
  animated types, 20
  declarative expressions, 21
  keyframes, 20
  messages, 20
general specification, 86
  action graph, 90
  model graph, 87
  time graph, 96
interaction constraints, 57
  virtual event, 60
  virtual port, 58
interaction-based program
  meta-scripts, 49
  scripts, 48
interaction-based programming, 36
  coordination activities, 47
  production activities, 39
  run-time evaluator, 52
interaction-based simulation, 31
multi-modeling, 66
rasp toolkit, 35
  application development, 34
reuse guidelines, 106
  behavioral, 112
  interactional, 107
  temporal, 116
script, 69
  arrangement, 48
  declarative, 72
  functional, 74
  kinds, 70
simulation
  methodologies
    constraint, 27
    declarative, 26
    functional, 27
    spatial, 27
  parts
    display builder, 24
    entities, 24
    executive logic, 24
    temporal management, 24
software composition, 77
  aggregation, 78
  assembly, 79
software interaction, 40
  attributes, 43
  time-varying, 68
unidirectional ports, 42
visual simulation, 1

240