INFORMATION TO USERS

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

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

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

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

**ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600**

UMI[®]

A Framework for Representing and Specifying Animation Models for Virtual Characters

By
Michael F. Valenta

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Doctor of Philosophy

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

March 12, 2001

© Copyright
2001, Michael F. Valenta

**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

**395 Wellington Street
Ottawa ON K1A 0N4
Canada**

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

**395, rue Wellington
Ottawa ON K1A 0N4
Canada**

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-60973-1

Canada

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis,

**A Framework for Representing and Specifying Animation
Models for Virtual Characters**

submitted by

Michael F. Valenta

Frank Dehne
(Director, School of Computer Science)

Will LaLonde
(Thesis Supervisor)

Brian Barry
(External Examiner)

Carleton University
March 12, 2001

Abstract

In this thesis, we are concerned with animation models for virtual characters. More specifically, we wish to promote the reuse of animation models and the components and relationships used to construct them both within a single character and between multiple characters. We also wish to provide the possibility for reuse between separate virtual character architectures. In order to accomplish this, we have provided both a low-level representation for animation models and a specification mechanism for building animation models for this representation.

We derived our model representation from the areas of Synergetics and Dynamic Pattern Theory. The resulting Virtual Behavior(ViB) model represents motion as a collection of time varying variables, or channels, that are evaluated at discrete time intervals. Each channel has a corresponding signal, a function of the model state, which is used to determine the change, or delta, in the variable since the previous evaluation time. A variable may be modified by several sources. Consequently, a channel's signal is made up of multiple independent contributions functions whose outputs are combined, or blended, to determine the output of the signal.

Animation models are typically specified at a higher-level of abstraction than that provided by the ViB model. Consequently, we have combined concepts from Alexander's work on Pattern Languages with some programming language scoping concepts in order to define pattern oriented composition. Using this technique, we have built several levels of abstraction that allow the specification of animation models using common animation model abstractions and the translation of these specifications into the ViB model representation. We have also defined a layered animation model organization that provides appropriate abstraction barriers between model components.

Acknowledgements

First of all, I would like to thank the members of my committee, Brian Barry, Jacques Raymond, Neal Holtz and Jörg-Rüdiger Sack for the time they spent reading my thesis and their questions and comments during my defense. Also, thanks to my supervisor, Wilf Lalonde, for his input, especially during the writing of this thesis. I would also like to thank John Pugh and Evangelos Kranakis for giving me the opportunity to teach at Carleton and Irwin Reichstein for his guidance and input during my first teaching assignments. Thanks also to Dwight Deugo for his interest and collaboration in my work.

Much thanks go to the Computer Science office staff at Carleton and more specifically to Marlene Wilson, Barb Coleman, Rosemary Carter, Maureen Sherman, Judith Srna, Nicki Enouy and Linda Pfeiffer. Their support and understanding was invaluable both in teaching and in completing this thesis. I am also grateful for the many fellow students that I have had the privilege of interacting with while at Carleton. Specifically, I would like to thank Jason Morrison for showing an interest in my research.

I would like to thank my family for their support. A special thanks to my parents, Noreen and Denny, who have always supported and inspired me and to my siblings, Kathy and Rick, whose success and good nature have been an inspiration. Also, thanks to my parents-in-law, Rudy and Henny Biel, who have always kept me in their prayers. Thanks to my son Daniel, who can't really do much yet but whose presence has given me a new perspective on life. But most of all, I would like to thank my wife, Krista, for both supporting me and continually pushing me to finish. Without her, I would never have completed this work.

Finally, I would like to thank God for his faithfulness during this time, not only giving me the strength to finish my thesis, but also providing me with a wife, a son and many other special people who have supported me in so many ways.

Contents

Abstract	iii
Acknowledgements	iv
1 Introduction	1
1.1 Motivation	2
1.2 Goal	5
1.3 Resulting Architecture	6
1.3.1 ViB	6
1.3.2 POCL	8
1.4 Contribution	10
1.4.1 ViB Model and Evaluation Engine	10
1.4.2 Pattern Oriented Composition	11
1.4.3 Abstractions as Pattern Libraries	11
1.4.4 Specification Reuse	12
1.5 Overview	13
2 Background	14
2.1 Models for Animation	14
2.1.1 Geometric Model	15
2.1.2 Kinematic Models	18
2.1.3 Physics-Based Modeling	21
2.1.4 Behavioral Modeling	24
2.1.5 Cognitive Modeling	27

2.2	Architectures for Animation	27
2.2.1	Subsumption Architecture	28
2.2.2	Artificial Fish	28
2.2.3	Hamsterdam	29
2.2.4	Improv	30
2.2.5	Humanoid	31
2.2.6	Others	31
2.2.7	Summary	32
2.3	Abstractions for Animation	33
2.3.1	Levels of Abstraction	33
2.3.2	Organizational Abstractions	35
2.3.3	Meta-Level Abstractions	37
2.4	Patterns	40
2.4.1	Model Representation	40
2.4.2	Model Specification	43
3	Virtual Behavior Model	47
3.1	Requirements	47
3.1.1	Simple	48
3.1.2	Flexible	48
3.1.3	Natural	48
3.2	The ViB Model	49
3.2.1	Channels	49
3.2.2	Channel Output	50
3.2.3	Channel Signal	50
3.2.4	Channel Contributions	51
3.2.5	Systems	52
3.3	ViB Model Entities	54
3.3.1	Channel Output	54
3.3.2	System and Channel	55
3.3.3	Signal	57

3.3.4	Function	60
3.4	ViB Engine	63
3.4.1	Interfacing to Other Architectures	63
3.4.2	Model Evaluation	65
3.4.3	Runtime Issues	66
3.5	Summary	67
4	Using Patterns for Specification	70
4.1	Requirements	70
4.1.1	Composition	70
4.1.2	Levels of Abstraction	71
4.1.3	Simple	72
4.2	Pattern Based Specification	72
4.2.1	Pattern	73
4.2.2	Entities	74
4.2.3	The Pattern Oriented Composition Language	75
4.3	Syntax	75
4.3.1	Entities	75
4.3.2	Pattern Configuration	76
4.3.3	Pattern Definition	77
4.3.4	Variable Reference	77
4.3.5	Pattern Primitive	78
4.3.6	Context	78
4.3.7	Pattern Library	79
4.4	Semantics	79
4.4.1	Pattern Resolution	80
4.4.2	Variable Scoping	82
4.4.3	Pattern Definitions as Entities	85
4.4.4	Pattern Signatures	86
4.4.5	Pattern Libraries	87
4.4.6	Entities and Pattern Primitives	96

4.5	POCL Tools	97
4.5.1	POCL Browser	97
4.5.2	POCL Compiler	99
4.5.3	Primitive Patterns	100
4.6	Summary	101
5	Specifying ViB Models as Patterns	103
5.1	Creating ViB Models	103
5.1.1	Entities	104
5.1.2	Primitive Patterns	105
5.1.3	Pattern Configuration	106
5.1.4	Pattern Definition	111
5.1.5	Pattern Libraries	115
5.2	Pattern Organization	117
5.2.1	Pattern Hierarchy	118
5.2.2	The ViB Library	123
5.2.3	The APE Library	126
5.3	Summary	131
6	Animation Model Specification	133
6.1	Low-Level Abstractions	134
6.1.1	Encapsulation	135
6.1.2	Channel Contribution Schemes	140
6.1.3	Variables	147
6.1.4	Systems with Variables	155
6.1.5	Other General Pattern Libraries	163
6.2	Animation Model Abstractions	166
6.2.1	Motion Model	168
6.2.2	Controller Model	169
6.2.3	Action Model	173
6.2.4	Behavioral Model	177
6.3	Summary	184

6.3.1	Low-Level Abstractions	184
6.3.2	High-Level Abstractions	185
7	Specifying a Virtual Character	186
7.1	Supporting Architecture	186
7.2	Character Model	188
7.3	Structural Model	192
7.4	State Feedback Model	193
7.5	Sensory Model	194
7.6	Motion Model	195
7.6.1	Kinematic Motion	197
7.6.2	Surface Contact	198
7.6.3	Surface Friction	200
7.6.4	Other Constraints	201
7.7	Controller Model	202
7.7.1	Action Control	202
7.7.2	Posture Control	205
7.8	Action Model	208
7.8.1	Action	209
7.8.2	Action Layering	210
7.8.3	Posture Actions	212
7.9	Behavioral Model	213
7.9.1	Desires	213
7.9.2	Sensory Information	214
7.9.3	Intention Generation	214
7.9.4	Action Sequencing	217
7.10	Results	222
7.10.1	Appearance	222
7.10.2	Motion	223
7.10.3	Behavior	224
7.10.4	Response Time	224

7.10.5 Design	225
7.10.6 Reuse	226
7.11 Summary	226
8 Conclusion	228
8.1 Summary	228
8.2 Contribution	230
8.3 Future Work	231
8.4 Conclusions	234
Bibliography	235
A Miscellaneous Metrics	243
A.1 Smalltalk Implementation	243
A.2 Primitive Patterns	243
A.3 Patterns and Libraries	244
A.4 ViB Model	244
B Detailed ViB Expansions	245
B.1 The ArticulatedObject Library	246
B.1.1 The Ship Library	248
B.1.2 The Action Library	253
B.1.3 Resulting ViB Model	255
C Pattern Libraries	257
C.1 POCL Library	257
C.2 ViB Library	258
C.3 AniMoS Library	260
C.3.1 ActivatedChannelContribution Library	262
C.3.2 WeightedChannelContribution Library	262
C.3.3 NormalizedChannelContribution Library	262
C.3.4 StrengthAdjustedChannelContribution Library	263
C.3.5 ImportanceAdjustedChannelContribution Library	263

C.3.6	ChannelVariable Library	263
C.3.7	ArticulatedVariable Library	264
C.3.8	Variable Library	264
C.3.9	MemoryVariable Library	264
C.3.10	BlendedVariable Library	265
C.3.11	EnablingVariable Library	265
C.3.12	SuppressibleVariable Library	266
C.3.13	InhibitatableVariable Library	266
C.3.14	ActivationVariable Library	266
C.3.15	DisplacementChannel Library	267
C.3.16	DeltaDisplacementChannel Library	267
C.3.17	VariableSystem Library	267
C.3.18	GradualActivation Library	268
C.3.19	ActivationOrdering Library	268
C.3.20	FiniteStateMachine Library	269
C.3.21	ControllerModel Library	271
C.3.22	ActionModel Library	273
C.3.23	BehavioralModel Library	276
C.3.24	Module Library	280
D	Selected Patterns of the APE Libraries	282
D.1	APE Library	282
D.1.1	ArticulatedObject Library	282
D.1.2	Character Library	282
D.1.3	Robby Library	287

List of Figures

1.1	ViB and POCL Architecture	6
3.1	Example System of Channels	53
3.2	ViB Engine Architecture	64
3.3	ViB Model Viewer	68
4.1	Example VRML Node Hierarchy	74
4.2	Resolution Context	85
4.3	Configuring a Library	88
4.4	Using a Pattern as Input	89
4.5	Lower Level Abstractions as Input	90
4.6	Library Containment	92
4.7	Library Extension	93
4.8	Dynamic Pattern Overriding	96
4.9	POCL Library Browser	98
5.1	channelOutput Expansion	107
5.2	ChannelContribution Expansion	108
5.3	Simple ViB Model for Kinematic Displacement	109
5.4	KinematicDisplacementChannel Definition	111
5.5	Using the KinematicDisplacementChannel Definition	112
5.6	KinematicDisplacementChannel Configuration Resolution	113
5.7	KinematicDisplacementChannel Library	116
5.8	Library Resolution	116

5.9	Pattern Hierarchy	118
5.10	Virtual Character Architecture	120
5.11	Primitive Pattern Libraries	121
5.12	Example Library Organization	122
5.13	The ViB Library	124
5.14	ViB Library Expansion	125
5.15	The APE Library	126
5.16	APE Library Expansion	128
5.17	The ArticulatedObject Library	129
5.18	Ship Library	130
6.1	Low-level Pattern Libraries	134
6.2	Expansion of System Configuration	139
6.3	Expansion involving NormalizedChannelContribution	147
6.4	Expansion of ChannelContribution	148
6.5	Outline of ChannelVariable	150
6.6	Expansion of a Variable Definition	152
6.7	VariableSystem Pattern	156
6.8	Example VariableSystem	156
6.9	CreateContextArray Resolution	158
6.10	Combining Variable Contexts	159
6.11	CreateContext Resolution	161
6.12	Types of Activation Signals	163
6.13	Activation Ordering	165
6.14	Animation Model Architecture	166
6.15	Higher-Level Pattern Libraries	168
6.16	Example Action Controller	171
6.17	Posture Model for a Flying Character	172
6.18	Flap Action	175
6.19	Action Layer for Posture Control	176
6.20	FocusOnObject Module	178

6.21 ApproachLocation Behavior Group	181
7.1 APE Architecture	187
7.2 Robby's Room	189
7.3 The ArticulatedObject Pattern	189
7.4 APE Pattern Libraries	190
7.5 AnimationModel Pattern Definition	191
7.6 Specification for the Character Robby	192
7.7 Robby's Structural Model	193
7.8 StateFeedbackContext Definition	194
7.9 Absorbing Collision Impact	200
7.10 Friction Causing Motion	201
7.11 Targeting Controller Dimension	203
7.12 TargetingController Definition	204
7.13 HeadTargetingController Definition	205
7.14 Looking at a Chair	206
7.15 Posture Model for Robby	206
7.16 The Moving Posture	207
7.17 Robby Having a Rest	208
7.18 Action Model of Robby	209
7.19 A Simplified Walk Action	210
7.20 The Locomotion Action Layer	211
7.21 Behavioral Model Layering	213
7.22 Intentions Model of Robby	215
7.23 Fatigued Action-Selection Group	216
7.24 Network to Approach an Object	218
7.25 ApproachObject Module	219
7.26 Approaching A Chair	220
7.27 Overriding Approach Object	221
7.28 ApproachTV Module	221
7.29 Approaching the TV	222

B.1	The ViB Library	245
B.2	The APE Library	246
B.3	The ArticulatedObject Library	246
B.4	Articulated Object Motion Model	247
B.5	InitialPositioning Definition	248
B.6	Ship Library	248
B.7	Resolution of APE Configuration	249
B.8	Ship Pattern Definition	250
B.9	Ship Pattern Resolution	250
B.10	AnimationModel Pattern Resolution	251
B.11	ActionModel Pattern Resolution	253
B.12	Action Pattern Resolution	255

Chapter 1

Introduction

With the growing popularity and accessibility of 3D applications and virtual environments, there is also a growing demand for virtual characters to inhabit these environments. A virtual character can be a simple character that performs actions under user control or a more complex character that exhibits autonomy. The goal of these characters is either to convey information to the user or to interact with the user in interesting or meaningful ways.

There are several aspects of these characters that are important in determining an observer's perception of a character. The *appearance* of the character plays a large role in defining the character. Creating realistic and interesting looking scenes and characters has long been the focus of computer graphics research and there are many available platforms for creating and displaying high quality images generated from these. With the recent boom in computer graphics hardware, the ability to display high quality computer images is available even on low end computers.

Another aspect of a character is its *motion*. Work in the area of computer animation focuses on generating a sequence of images of a character that can be displayed in a particular order and results in recognizable movements such as walking or swimming. Techniques for defining motion range from using prerecorded positions, or key-frames, to defining models based on dynamics that generate coordinated motions that adhere to the laws of physics.

The final aspect of a character is its *behavior*. The basic motions of a character

can be viewed as low-level control corresponding to repeatable motor skills or actions. Behavior is high-level control and involves a sequencing of lower-level motions that convey intention. For instance, a character may walk over to a chair and sit down if it is tired or hunt about its environment for food if it is hungry. Creating behavior models has been an active area of research in both computer animation and artificial intelligence.

1.1 Motivation

The representation of a virtual character is typically separated into two models: the *structural* or *geometric* model and the *motion control* or *animation* model. The geometric model defines the appearance of the character and is typically controlled through a set of parameters that allow modification to certain aspects of geometry. The animation model defines how these parameters are manipulated over time in order to animate the virtual character.

Animation models can be categorized by the type of control they provide, be it kinematic, physical, behavioral or cognitive [45, 66]. Each model has certain strengths and weaknesses. Kinematic models have low computation cost and can incorporate recorded data from live subjects thus producing natural looking behavior. However, the resulting animation is only applicable to a small number of situations. Physical models produce realistic looking motion that can react to environmental conditions but rely on complex calculations. Behavioral models produce interactive behaviors that convey autonomy but are difficult to create and typically rely on kinematic or physical models to provide lower level motor skills. Cognitive models contain knowledge and planning capabilities and, like behavioral models, typically rely on lower-level models.

A related area of research is concerned with the development of architectures that support interactive, autonomous virtual creatures [11, 7, 71, 53]. In order to provide behavior that conveys autonomy, architectures for autonomous creatures typically rely on multiple animation models that provide different levels of behavior. For instance, lower level behaviors, such as walking or swimming motor skills, may be represented

using a kinematic or physical model while higher level behaviors, such as decision making or goal determination, may be represented using a behavioral model. Some of the proposed architectures for autonomous creatures focus on providing lower level models that are suitable for use by a higher level control system such as a behavioral model or an agent based system [36, 14]. An important aspect of these architectures is the interface between levels as these interfaces dictate how much control a higher level can exhibit over a lower level.

Existing research [71, 11, 55, 36] in the area of animation models and autonomous creature behavior has concentrated on defining overall system architectures and the components and models that make up these architectures in a manner that is specific to each of the proposed architectures. Typically, the only reuse between architectures and the models they incorporate is in the form of ideas that are borrowed or refined from other work. However, there are several common aspects between many of these models. For instance:

- Models typically consist of a collection of atomic components that interact in some manner to determine the overall behavior exhibited by the character. These components may be referred to as motions, motor skills, actions or behaviors depending on the level of behavior being modeled.
- Many models involve connecting components together in a layered, hierarchical, tree or network structure.
- The effects of several different components are often combined or blended together.
- Information is communicated between the components of a model or to higher or lower level models as numeric values, symbols or data structures.
- Input to a model, either from a higher level model, sensors or the user, is processed by the model in order to determine the output which is sent to either a lower level model or effectors connected to the parameters of the geometric model.

Given these similarities, it would appear that creating reusable models is possible.

However, reuse of models and components from current architectures is complicated by several factors.

- The relationships between components are often hard-coded into the model's underlying architecture. This simplifies the creation of components for use by the model but does not allow for components that require special handling. Special handling identified by the original developer may have been implemented as part of the architecture but the ability to add unforeseen special handling does not often exist. Every time a new model architecture is developed, the underlying component relationships must be re-implemented specifically for the architecture.
- The interfaces between models have small bandwidths. Again, this simplifies the creation of components for a model but leads to inflexibility when higher level models cannot perform properly due to shortcomings in the lower levels they depend on. The ability of some components in a model to exercise more control over lower level models may be beneficial in some cases.
- Often models concentrate on a particular type of behavior selection, whether it is based on finite state machines [17], action-selection networks [10] or some other method [54]. However, they do not often provide a means to combine these different types of selection mechanisms in a single network. This type of interfacing between models may be helpful in promoting the reuse of animation models between creatures of similar structure or function.
- The specification methods for different models are often tailored to the model itself and do not lend themselves to be reused by others. Also, components created for a specific model are not typically made available for use by others. There are commercially available animation models and animation model components, especially for human characters, but these are usually specific to a particular architecture or evaluation engine and are not easily integrated with other animation models.

1.2 Goal

The goal of this thesis is to define a framework that can not only be used to provide animation models for virtual characters but also will allow animation models and the elements that constitute them to be more readily reused and shared. We use the term element to refer not only to the components that are typically thought of or specified as atomic actions or behaviors, such as walk or find food, but also, the meta-behaviors that describe or define relationships between components, such as component interaction, selection, combination and blending.

Reuse of animation model elements can occur at different levels. Animation models for a particular character could be reused in different architectures or applications. Individual components or even a set of components could be reused between similar characters. Component relationships such as interconnection or blending schemes can be reused in different layers of the same animation model. Our aim is to promote the ability to specify and represent animation models in such a way that they can be reused and shared. Therefore, we do not wish to promote or define a particular representation for actions and behaviors, or to define a particular style or set of styles for action-selection or action combination. Instead we wish to provide a model that is capable of representing different types of animation model components and organizations. Also, we wish to specify these animation model elements in such a way that the specifications can be reused with different characters and models.

In addition to promoting the reuse of animation model elements between different characters and architectures, a further goal is to support the creation of animation models that can incorporate kinematic and physical notions in addition to supporting behavioral models. Our aim is not to replace or compete with models that are tightly coupled to a particular geometric model in order to provide higher motion fidelity, such as physics-based models. Instead, we wish to provide a framework that can make use of such lower-level models if they are available while being able to provide or at least approximate the necessary motion control when they are not.

1.3 Resulting Architecture

In order to meet our goals for the promotion of animation model element reuse, we have proposed and implemented the architecture shown in Figure 1.1 [77, 76]. This architecture consist of two components. The Virtual Behavior (ViB) component is an engine for executing the animation models of virtual characters while the Pattern Oriented Composition Language (POCL) component provides the specification environment for building animation models.

Figure 1.1: ViB and POCL Architecture

1.3.1 ViB

The Virtual Behavior (ViB) architecture consists of the following components.

- A low-level ViB model which represents animation models as a set of variables referred to as channels that are grouped and organized hierarchically in systems. Channel output is determined by combining, or blending, the output of a set of contribution functions.
- A ViB engine which coordinates the evaluation of a ViB model with other architectures and provides connections to these architectures in order to allow the output of a ViB model to be used to animate virtual characters and to allow

information from the virtual character's environment to be fed back into the ViB model.

- A ViB model viewer which allows the inspection of the channels of a ViB model at run-time for the purposes of debugging.

One of the main strengths of the ViB model is that the low-level of representation and small number of concepts that constitute this architecture leads to an uncomplicated and compact implementation. Our implementation of the ViB engine consists of in the order of 10 structural object classes for model representation and evaluation and approximately 50 types of functions.

However, the success of the model hinges on its ability to represent higher level animation model concepts. The model was derived from Synergetics [34] and Dynamic Pattern Theory [38] which are disciplines that describe observed behavior patterns in terms of their dynamics. We have adapted some of the concepts from these fields to create a discrete time-step computational model that approximates dynamics using delta values. Thus the ViB model is capable of representing reactive kinematic and behavioral models. Although not capable of true physics-based modeling, it is also capable of incorporating physical notions but the success of these types of motion depends on the frame rate of the animation.

Another drawback of the model is that, as the number of components and component interactions that constitute a model grow, so do the number of channels and channel contribution functions that need to be evaluated on each tick. This reduces the scalability of the model. We have improved this through the use of system structuring and evaluation optimizations that allow the disabling of systems in the model that are not contributing to behavior determination.

Because of these factors, we feel that the ViB engine is a good candidate for use as a behavioral virtual machine that other architectures can connect to in order to add behavioral animation capabilities. Also, because of its compact implementation, it could be easily built into other virtual character architectures.

1.3.2 POCL

One of the limitations of the ViB model representation is that it is difficult to build complex animation models directly. Animation models are typically specified at a high level of abstraction using entities such as controllers, actions and behaviors while the ViB model representation consists of functions, channels and systems. We propose a technique we refer to as pattern oriented composition (POC) as a means of bridging the gap between these high level abstractions and our low level ViB model representation. We refer to the associated specification language as the Pattern Oriented Composition Language (POCL).

Our inspiration for pattern oriented composition comes from Alexander's work on patterns of architectural design [2]. In his work, Alexander defines a pattern as a description of a relationship between a context, a set of forces and a configuration or arrangement of a set of entities. We have adapted this to entity composition by viewing a pattern as a definition of a relationship between a set entities configured in a particular context.

Key to both Alexander's and our interpretation of patterns is the idea that

1. the configuration of a pattern includes both entities and smaller-scale patterns that help fill in the details of the relationship the pattern represents, and
2. the context of a pattern is determined by larger-scale patterns that are being used.

Therefore, pattern oriented composition is a hierarchical process where larger-scale patterns provide the context for smaller-scale patterns by defining which entities and still smaller-scale patterns should be configured by the pattern.

Larger scale patterns in this process act to define the higher-level model relationships such as the type of motion control or action-selection mechanisms used while smaller-scale patterns act to define the effects of a particular entity, such as an action, on the underlying ViB model. The smallest-scale patterns in the pattern composition hierarchy are primitive patterns that use one or more parameters from

the context to create an entity representing an object or relationship between objects in the underlying model.

In essence, patterns are similar to macros in Lisp [65] and the conversion, or resolution, of pattern definitions into the corresponding ViB model representation is a process similar to macro expansion. However, pattern oriented composition predominantly uses dynamic scoping to determine the entities and smaller-scale patterns that a pattern configures. In order to provide control over dynamic scoping during composition, patterns are organized in a pattern library hierarchy that provides static relationships and also acts to promote reuse through the use of inheritance-like relationships.

The POCL environment, as shown in Figure 1.1, supports pattern oriented composition and consists of the following components.

- The pattern library hierarchy which contains pattern definitions representing the relationships between the entities used in constructing animation models and virtual characters.
- The POCL editor which allows the specification of pattern libraries and definitions along with the browsing of the pattern library hierarchy. Cross referencing tools are provided to aid designers in pattern creation.
- The POCL compiler which translates patterns for animation models into the corresponding ViB model in order that the model can be used by the ViB engine to animate a character.

As with the ViB evaluation engine, the implementation of the POCL compiler is simple and compact. Our implementation consists of approximately 15 object classes. The compiler simplicity is due to the fact that the compiler deals strictly with the composition of entities and not with the specifics of the entities themselves. POCL also provides approximately 20 primitive patterns to support general composition and concise specification.

The number of primitives required for a particular target architecture depends on the underlying model. For the ViB model, we have provided approximately 10

pattern primitives, most of which correspond to contribution functions. We have also provided pattern primitives for building character geometries using a VRML-like node hierarchy and for connecting ViB model output to these geometries.

Using these primitives as a base, we have built a large pattern library that can be used to generate virtual character animation models using common animation model abstractions. This was accomplished by providing and connecting several different layers or levels of abstraction. These layers include schemes for the low-level weighted combination of functions and the communication of information, common animation model abstractions such as controllers, actions and behaviors, and example character models.

1.4 Contribution

Our work consists of three main components: the ViB model and supporting architecture, the POCL specification environment for pattern oriented composition and the pattern libraries implemented using POCL to map models specified using higher-level abstractions to the ViB model representation. The contributions of these components are summarized below.

1.4.1 ViB Model and Evaluation Engine

From the standpoint of architectures for computer animation and virtual reality, our contribution is the definition of a simple ViB model representation that is capable of representing a wide variety of animation models while, at the same time, is easily implemented and interfaced to other architectures. This makes the representation and the supporting evaluation engine a prime candidate for implementation as a virtual machine that could be interfaced to, or implemented as part of, different architectures to provide them with behavioral capabilities and allow animation models to be shared between them.

1.4.2 Pattern Oriented Composition

Using POCL, we will illustrate how dynamic scoping can be used in conjunction with a static pattern library structure to provide a useful method for building low-level ViB models using high level abstractions. By incorporating hierarchical scoping concepts similar to inheritance from object-oriented programming, we have improved the ability to reuse pattern definitions and libraries. We have also applied this technique to the generation of simple geometric models using a node structure and syntax similar to VRML. Finally, because POCL deals solely with composition, the implementation of the supporting architecture is simple and compact.

1.4.3 Abstractions as Pattern Libraries

Our main contribution in animation model specification is not the POCL environment itself but is, instead, the animation model specification pattern libraries that are implemented using POCL. We have implemented a comprehensive library of abstractions for creating animation models and have used these libraries to generate ViB models that provide kinematic and behavioral level motion control. By using these libraries, a designer is able to specify model entities at a high level, be it an action-selection hierarchy or a network of interconnected nodes, and generate the corresponding ViB model from these specifications.

Using POCL, we have designed a pattern library structure that provides several different layers of abstraction for mapping higher-level animation model abstractions to ViB models, from low-level weighted combination schemes to high-level organization schemes. Because the different layers required to map high-level abstractions to the ViB model representation are implemented as pattern libraries, the designer has the option to modify the abstractions represented by intermediate layers within the same specification environment and can also provide as large or as small a bandwidth between layers as required. It is for this reason that we strived to provide only the lowest level abstractions necessary to build ViB models as primitive patterns in POCL and built all other layers of abstraction on top of these using patterns and libraries.

As for the animation model abstractions themselves, most are based on or adapted

from previous work in the field. Some variations are due to the fact that we are mapping them to the ViB model representation instead of the evaluation model provided by typical programming environments. The contribution of this thesis is to illustrate how POCL can be used to specify animation model entities at a high level while allowing these specifications to be used to generate the corresponding low-level ViB model. This also acts to illustrate that the ViB model can indeed be used to provide complicated kinematic and behavioral motion control models.

1.4.4 Specification Reuse

Although we have achieved specification reuse through the creation of lower-level abstractions that are common between several higher-level animation model abstractions, this is no different than the reuse that would be achieved had we used an object-oriented programming language. However, we feel that there is also potential for reuse between different architectures. This can be accomplished in two ways. One approach is to connect a common ViB engine implementation, such as the one we created for this thesis, to different architectures thus allowing these architectures to share animation models for similar virtual characters. The POCL pattern library browser and ViB model viewer could be used to aid in the tailoring of model specifications required to handle any differences between the characters or architectures involved. Because both the ViB engine and POCL compiler are easily implemented, a second approach is to provide an architecture that contains integrated implementations of these components. In either of these approaches, pattern definitions and libraries can be shared between designers in order to provide reuse.

Another possible avenue for specification reuse is to use pattern libraries to provide layers or levels of abstraction within a specification. Conceptually, higher-level abstractions specified using patterns could be mapped to other underlying animation model representations by providing the necessary pattern primitives and lower-level pattern libraries to replace those related to the ViB model.

1.5 Overview

In Chapter 2, we review previous work in computer animation, including animation models and animation architectures. We also describe some of the common abstractions that are used when implementing animation models and present some of the work from which we derived our ideas about patterns and how they can be used to represent and specify animation model elements.

In Chapter 3, we present the ViB model we use for representing animation models. In addition, we look at the evaluation engine and other supporting tools that aid in the creation and use of ViB models. ViB models are built using patterns and pattern libraries that are specified using POCL whose syntax and semantics are presented in Chapter 4. In Chapter 5, we tie the previous two chapters together by introducing the ViB pattern primitives and describing the high-level library structure that supports the construction of ViB models and virtual characters.

Chapter 6 presents the patterns and pattern libraries we implemented in order to support high level specification of animation model abstractions and conversion of these specifications to the underlying ViB model representation. How these pattern libraries are used to create animation models for virtual characters is illustrated in Chapter 7. Then, we present our conclusions along with possible future research in Chapter 8.

Chapter 2

Background

Our interest lies in creating virtual characters that interact with their environment. More specifically, we wish to simplify the creation of characters by promoting the reuse of the models that control how the characters act. To determine how to best accomplish this, we need to look at previous work in the area.

First we will survey the different models that are commonly used to represent different aspects of virtual characters. Then we will describe some of the projects and architectures that have incorporated these models in order to create virtual characters that exhibit interactive behavior. From these, we will identify and describe common abstractions that are used to build animation models for virtual characters. We will then discuss two areas of research dealing with the concept of patterns from which we have derived our approach for animating virtual characters and promoting reuse of animation model elements.

2.1 Models for Animation

It is often difficult to categorize the various models that are used to generate computer animation due to the fact that many models are either a combination of, rely on, or overlap with other models. An early classification scheme proposed by Zeltzer [86] classifies the representation and control of animation as occurring at the level of guiding, programming or tasks. Guiding level animation systems require the animator

to specify the geometric data required to produce the animation. Procedural systems reduce the amount of animator specification by procedurally generating the geometric data while still providing a smaller set of parameters to allow animator control. In task level animation, a character is controlled through high level commands such as “Walk over there”.

Magnenat-Thalmann and Thalmann [45] proposed a scheme for classifying animation models, or *motion control methods*, by considering the type of information that the method uses in order to control the motion. Methods classified as providing *local control* use geometric data, such as positions or joint angles and include techniques such as parametric key-framing and motion capture. Local methods rely on the animator providing a large amount of geometric data in order to specify animations. *Globally controlled* methods make use of kinematic and dynamic information about the objects being animated in order to simulate the physical laws of motion. The third category is *behavioral animation* which makes use of the relationships between an object and other objects and conditions in the environment in order to determine how the object should act.

Funge [28] describes a similar classification scheme to the one above. The *geometric model* provides the shapes and transforms that represent the object being animated. *Kinematic models* and *physical models* correspond to local control and global methods of Magnenat-Thalmann’s classification scheme respectively. In his scheme, Funge separates the behavioral animation method into *behavioral models* that react to environmental conditions and *cognitive models* that use knowledge to perform deliberate actions. We use this categorization scheme for describing animation models.

2.1.1 Geometric Model

The structure of an object is typically represented by a set of points and a set of relationships between these points. The points that define an object are contained or specified in a coordinate system. The relationships between points typically define surfaces such as lines, polygons, or other geometric shapes. These shapes are often

further described by other attributes such as color and texture.

Since the geometric shape of an object is derived from a set of points, controlling the motion of these points is fundamental. At the lowest level, an object can be animated by changing the location of one or more of these points in their respective coordinate systems. However, since it is usually the case that there are fixed spatial relationships between the points, they are often grouped together in such a way that their locations can be controlled as a unit. Hence, such a group of points is said to have a local coordinate system C_l and may also have an associated transformation, T , that is used to determine the location of the points in another coordinate system C_w .

Rigid Bodies

The simplest type of geometric model to animate is a rigid body. In a rigid body, the relationships between the points that define it are constrained such that the shape of the body doesn't change. In other words, the points that represent the body are at fixed positions relative to the local coordinate system C_l of the entire body.

The Chasles' theorem states that any displacement of a rigid body is equivalent to the sum of a translation of any one point on that body and a rotation of the body about an axis through that point. Therefore, the location of a rigid body in an external coordinate system can be described by a homogeneous transformation [25] representing a translation followed by a rotation. Therefore, given the set of points $\mathbf{p}$ that defines the body, the position of the body in the external coordinate system C is $\mathbf{p}' = \mathbf{T}_w^l \mathbf{p}$ where $\mathbf{T}_w^l$ is the transformation equivalent to a translation from the origin of C_l to the bodies position in C_w and a rotation about axis passing through the point equivalent to the translated origin of C_l .

Non-Rigid Bodies

Like rigid bodies, non-rigid bodies are typically modeled such that the points that define the body are located relative to C_l . Therefore, positioning a non-rigid body in C is the same as positioning a rigid body. However, the relationship between points

that define a non-rigid body are not fixed. In other words, some or all of the points that define a non-rigid body are able change position relative to C_l .

One way of representing this is to divide $\mathbf{p}$ into subsets in which the local coordinate system of a particular subset may be positioned relative to the local coordinate system of another. This is the basis for articulated bodies that are discussed in the next section. The locations of the points that define a deformable body [68, 67] are typically constrained in some manner to ensure that the general shape of the object is maintained. For example, in [72, 71], a spring-model is used to determine the distances between a set of interconnected points that constitute the model of a fish. Similar models are used in [49] to represent snakes and worms

Articulated Bodies

Articulated bodies are a combination of open kinematic chains where the *base* of the chain is fixed and the *tip* or *end-effector* is free to move. A chain is viewed as a set of rigid bodies, referred to as *links*, connected together by *joints*, which constrain the motion of the links they connect. That is not to say that the components of an articulated figure need to be rigid. It is common to model the skeletal system of a creature using an articulated body and use this skeleton to control deformable sub-models in order to produce more realistic looking creatures [73, 61, 37].

One way to describe articulated body motion is by using Denavit-Hartenberg notation [22]. In this notation, the links and joints are arranged in a chain consisting of n joints where link k ($0 \leq k \leq n$) is said to connect joint k to joint $k + 1$. The link represents the physical connection between joints but in the notation itself, the relationship between joint k and $k + 1$, including the displacement associated with link k is captured in the parameters of joint $k + 1$. Therefore, the joint parameters are sufficient to describe motion within the body.

When using Denavit-Hartenberg parameters to define articulated bodies, a joint is either translational (prismatic) or rotational (revolute). The angle of rotational joint k at time t is given by $\theta_k(t)$ while scalar displacement of a translational joint k is given by $d_k(t)$. When using articulated bodies to represent the skeletons of biological creatures, translation joints are not usually considered and multiple rotational

degrees of freedom are often combined into a single joint and controlled by separate parameters. For instance, in [44], a joint is said to have up to three rotation degrees of freedom referred to as flexing, pivot and twisting. Therefore, angles associated with this type of joint is given by $\theta_k(t) = (\theta_k^{flexion}(t), \theta_k^{pivot}(t), \theta_k^{twist}(t))$.

In an articulated body, each joint provides the local coordinate system for its down-chain (closer to the tip) link and any joints connected to that link. Associated with joint k is a transformation, $\mathbf{T}_{k-1}^k(\theta_k(t))$ that defines the local coordinate system of the joint relative to joint $k - 1$ as a function of the joint angles. The coordinate frame of the joint relative to the external coordinate frame of the articulated body is then given by $\mathbf{T}_w^l \mathbf{T}_0^1(\theta_1(t)) \mathbf{T}_1^2(\theta_2(t)) \dots \mathbf{T}_{k-1}^k(\theta_k(t))$.

An articulated body can be a single chain of links connected by joints, such as a robot arm or a snake but more typically consists of a combination of multiple chains where some joints have more than one child joint. This leads to a hierarchical arrangement of joints with a single root or base joint with potentially multiple branches from the joints in the hierarchy. Bodies that are direct children of a joint, including other joints, are positioned relative to the parent joint's coordinate system.

2.1.2 Kinematic Models

In kinematic models, motion is represented at the geometric level and is controlled by varying positions of objects, angles of joints or other parameters of the model without regard for any information that is not available or representable at the geometric level. Generally, kinematic models have low computation cost due to the simple calculations used to determine the values of the geometric parameters. Furthermore, the animator typically has a high degree of control over the resulting motion and forward kinematic models can be created easily using applications that support forward kinematic methods such as key-framing or procedural motion specification.

However, building kinematic models requires a large amount of specification on the part of the animator and the resulting animation is not robust in the sense that its applicability is limited to a narrow range of situations. Application to new situations often results in discontinuities or irregularities in the animation. Inverse kinematic

methods can be used to reduce animation specification and improve robustness but, at the same time, reduce the control an animator has over the resulting motion.

Forward Kinematics

With forward or direct kinematics, the values of the parameters that control a geometry are either provided directly or calculated from velocities or accelerations. For articulated figures, the parameters of interest are the joint angles that configure the figure and hence result in a positioning of the end-effectors. The calculations involved are straightforward using the methods described in Section 2.1.1. However, controlling the position of end-effectors using forward kinematics is difficult.

Key-Framing

Key-framing is a type of forward kinematics that relies heavily on the animator to supply specific values for the parameters that control the geometry being animated. For articulated figures, parametric key-framing is often used and involves specifying the angles of the joints at various times and then having the system fill in missing values for other times using some form of interpolation. A related method is to capture motion data using special equipment that observes live subjects and then to use this data to generate the key-frames used to animate the geometric model.

The quality of animation produced using key-framing is a function of the animator and the tools used to create the animation. In combination with motion capture, the animation produced can be natural, realistic and of high quality. However, this type of animation is highly tailored to a particular geometric model and a particular situation. Applying a key-framed sequence to different models or different situations often reduces the quality of the animation considerably and may cause interpenetrations or other irregularities. Consequently, this technique is more suited for creating animated motion sequences involving highly controlled environments such as those used for videos and film. Work has been done to combine key-framing with other techniques in order to map sequences to objects of different shape and scale [30] and to generalize sequences to situations that differ slightly from those for which the

animation was initially created [84].

Procedural

In procedural methods, the animation is described by an algorithm or formula that relies on kinematic information to determine the values to be assigned to the control parameters of the geometric model. One means of procedural modeling involves creating motion control procedures in a general purpose language. There have been many instances of this type of approach. For instance, in [59], extensions were added to Lisp to create ASAS, a general purpose programming language for creating computer animation. More recently, May developed the procedural animation language AL [48], an extension of scheme, which he used to define the skeletal structure for research on anatomy based modeling of muscles [61]. Others provide specially designed language or notation to create motion procedures; which is the approach taken for LifeForms [21, 19].

Another means of procedural animation is to associate functions with one of more parameters. The TWIXT system [32] uses functions attached to tracks representing the parameters of an object to control the object's motion. Perlin [54] uses a similar approach in the IMPROV system to create groups of functions representing motions. Others use Fourier principles to incorporate secondary motion into a track [20, 75].

Procedural techniques often require individual procedures to have exclusive access to the geometric parameters they affect. Work has been done to blend the effects of multiple procedures [54, 14]. Blending can be in the form of sequential transitioning between procedures and parallel execution of procedures that control common parameters. When blending effects, special care must be taken to ensure that motions combine together properly.

Sophisticated procedural techniques can also be used to improve the robustness of other kinematic methods. However, these techniques still suffer from a lack of physical realism. Given a suitably slow motion or high sampling rate, the motion can also incorporate simple physics notions [56, 41]. However, higher speed motions and motions involving a high degree of interactions between entities will often result in motion that noticeably violates the laws of physics.

Inverse Kinematics

Inverse kinematic models simplify animation specification by using provided end-effector positions, orientations or trajectories to determine the required joint angles that will place an end-effector at the desired location. This reduces the number of parameters that must be specified in order to control a geometric model. However, there is no general closed-form solution for determining the joint angles given an end-effector position. Also, it is possible to have more than one valid configuration of joints that will place the end-effector in the desired position and it is also possible to have no valid configuration that satisfy the request.

In special cases, it is possible to constrain the problem in order to obtain a closed-form solution. The most common technique [29, 15] relies on the desired position of the end-effector being close to its current position. It uses a pseudo-inverse of the Jacobian matrix that describes the state of the joints along with the desired change in the end-effector position during a small time-step to approximate the required joint angles. Thus, an end-effector can be made to follow a predetermined path.

Inverse kinematic techniques rely on the animator or some other source to provide the motion paths of the end-effectors. Consequently, as with other kinematic methods, the specified animation is often dependant on the shape and scale of the geometry. Furthermore, although the amount of specification required by the animator is reduced, so is the control the animator has over the motion of the individual joints in the chain. The fact that there may be multiple valid configurations that place the end-effector in the desired position has been used in several cases [29, 15] to develop techniques for finding those configurations that may be more desirable due to other constraints, thus providing the animator with some additional control. Inverse Kinematics are also often used to adjust or correct motion derived using other kinematic techniques so as to improve robustness [84, 42].

2.1.3 Physics-Based Modeling

Physics-based modeling uses the dynamic equations of physics to determine object motion. Consequently, the forces and torques are the main properties that control the

motion. The difficulty in implementing physics-based models stems from Newton's third law of motion that states that for every force exerted on one object by another, there is an equal and opposite force on the second object. Therefore, the forces and torques acting on a set of interconnected or interacting objects, such as an articulated figure or a deformable mesh, are highly interdependent and must be considered in a single system of equations requiring the use of numerical integration and matrix optimization techniques [81].

Physics-based models are capable of creating animation that cannot be matched using kinematic methods. The degree of realism reached depends on the fidelity of the geometric and dynamic model, or, in other words, how well the models reflect the animal being simulated. Obviously, as the fidelity of a model increases, so does its structural and computational complexity. So, it is desirable to create models that are optimized for a particular purpose thus reducing the complexity as much as possible. However, this leads to dynamic models that are highly dependant on the geometric model they are animating and therefore not generally reusable. Given that the creation of these models is a complex process, this leads to limited availability of models.

Another consideration with physical modeling is interactions with objects that are external to the system of equations, such as those caused by collisions and surface contact. These interactions often cause discontinuities and instability in the evaluation of the system of equations especially in the case of large changes in acceleration that are caused by collisions. These situations can be handled in a few different ways. The simplest approach is to implement motion controllers that avoid collisions and contact altogether. In those cases where this is not possible, special techniques can be used to reset the system of equations after contact.

Physics-based techniques can generally be divided into two categories: *constraint-based* techniques that rely on kinematic constraints to control the motion and non-constraint-based techniques that rely on controllers to control the motion. Another way to differentiate between the techniques is to consider how the forces and torques are determined. Methods based on *forward dynamics* use supplied forces and torques to determine the accelerations and, hence, the positions and velocities of the objects

involved while *inverse dynamics* uses prescribed motion paths on one or more objects to determine the forces and torques required to generate the desired motion.

Constraint Based Methods

Constraint based methods rely on some form of kinematic constraints to control the motion. For instance, in inverse dynamics [35, 40], the desired accelerations of end-effectors are used to calculate the forces and torques required to generate the motion. This is of particular interest in robotics since the desired path for an end-effector can be used to calculate the required commands to the joint motors. It can also be used to generate physically plausible motion for articulated and deformable bodies but the resulting motion, however physically real, may not appear to be natural [71].

Other constraint based methods, such as spacetime constraints [83, 51], allow the specification of different types of constraints and rely on constraint optimization using iterative numeric techniques to determine the forces and torques required to give an optimal solution. Constraints can also be combined with kinematically generated motion in order to produce natural looking physics-based motion [57]. These methods offer the animator a fair amount of control over the animation but are computationally expensive.

Forward Dynamics

In models based on forward dynamics, the forces and torques are used to determine the positions, velocities and accelerations of the end-effectors or objects being animated. The motion is controlled by varying the forces and torques of the model either directly or indirectly. For instance, in [71], a set of interconnected mass-spring models are used to simulate a deformable fish. The motion of the model is controlled by varying the rest-length of the springs which in turn results in changes to the forces of the model. Brogan uses a similar approach to model one-legged hopping robots [16].

Controllers

One deficiency with physics-based models stems from the fact that the animator has little control over the resulting motion. It is difficult to create meaningful motion that resembles the motion of an animal being simulated by directly providing forces and torques. Motion can be controlled using constraints such as end-effector paths, which does provide the animator with some control but the more control exerted by the animator in this manner, the less likely the resulting animation will be physically realistic. Physics-based models are more typically controlled using controllers, or control procedures, that generate appropriate time-varying model input to perform certain repeatable actions such as moving forward and turning. Controllers can either be hand-crafted or generated.

Hand-crafted controllers are used in both [49, 72] to control snakes and fishes and in [46] to animate the upper body of a dancing human. Examples of hand-crafted controllers for human gait can be found in [18, 85]. Hand-crafting reasonable controllers is a difficult task. When simulating animals, information about how the animal moves can be used to aid this process. Controllers can be generated using optimization criteria along with iterative optimization techniques [79, 64, 78, 33]. It is also possible to generate controllers from a set of objectives for the motion, such as beginning and ending key-frames or end-effector positions [83, 80].

2.1.4 Behavioral Modeling

Behavioral models consist of entities that behave or react in response to their perception of the surrounding environment. In his seminal work on behavioral animation, Reynolds [60] simulated a flock of birds using a technique derived from particle systems [58]. Each “boid” perceives information about the proximity and velocity of other nearby “boids” and its reaction to the perceived conditions is determined by a small set of behaviors that use this information to try and stay close to and match the velocity of other nearby “boids” while avoiding collisions. The trajectories of each “boid” are not predetermined but, instead, are determined by the interactions between multiple “boids” and the behaviors they contain. The result is an emergent

flocking behavior.

Other early work in this area includes the work of Wilhelms [82] who proposed a system based on a network of sensors and effectors. More recently, behavioral models have been used to provide higher level behaviors for artificial life and autonomous agents. Tu [72, 71] uses a behavioral model to provide a set of behavior routines for artificial fish while Blumberg [11] incorporates a certain degree of learning into hierarchical behavioral networks.

An important consideration in behavioral modeling is determining how a character interacts with its environment and with the user. This can be broken down into different subcategories including how the character senses the environment, how the character affects the environment, how the character interacts with the user and how the model translates perception into action.

Perception

An important aspect of a behavioral model is how the actor perceives the surrounding environment [69]. In real creatures, perception is provided by the five senses. It is possible to try and directly simulate some of the senses for virtual characters [70, 52]. For robots, real sensors are required along with computational models that translate sensor data into a form usable by the behavioral model. However, computer animated creatures can make use of virtual sensors that query geometric information about the environment directly. This simplifies sensor creation but may not lead to behaviors that emulate those of real creatures. Consequently, some work uses a combination of simulated vision using image analysis and direct information retrieval [71].

Motion

Related to how an actor perceives the environment is how the actor affects it. As we saw in Section 2.1.1, an actor can be represented in an environment by a geometric model that is controlled by a set of parameters. Consequently, a behavior can control and animate the geometric model by modifying the values of the parameters that control it over time. More generally, we can say that the behaviors of an actor control

the values of a set of variables, some of which may directly or indirectly control the shape or location of the geometric model. We say indirectly because it is often the case that behavioral models do not directly control the parameters of the geometry (e.g. joint angles) but instead, provide parameters to an intermediate model such a kinematic or physics-based model.

Higher-level Control

Behavioral methods transfer control from the animator to the model itself. The motion of a creature occurs in response to the state of the model and the state of the environment as perceived by the model. However, higher-level control is often desirable as well. This type of control can be provided through additional sensors that allows input from a higher-level control system or a user [11] or through an additional mechanism that translates commands, either in a special format or natural language, into effects on the state of the model [6].

Behavior Arbitration

An actor typically consists of many behaviors, some of which may be complimentary, but inevitable, some of which are contradictory. Even behaviors that are conceptually complimentary may interfere with each other if their effects overlap. Consequently, both the combination of effects of complimentary behaviors and arbitration between competing behaviors must be considered.

The main techniques for merging the effects of multiple complimentary behaviors is to combine the change to common variables using some sort of blending scheme, usually involving weights. Other schemes may rely on using priorities to grant mutual exclusion to one of the overlapping behaviors, but this can be viewed as a specialized weighting. Effects blending is not usually considered part of a behavioral model but is instead typically implemented as part of a lower-level kinematic model that forms an abstraction layer between a behavioral system and the geometric model [54, 14].

For competing behaviors, the process of determining which behaviors are active is referred to as an *action-selection* mechanism. This area is an active area of research

especially in the artificial life and autonomous agent areas. Tyrell [74] defines the term action to refer to one of a set of mutually exclusive tasks where the demands of an action on the effectors of the animal are such that only one may be executed at any one time. The task of action selection is choosing the most appropriate action to perform. One of the main techniques used to perform action-selection in behavior networks involves spreading activation [43, 74, 11]. An alternative approach is energy minimization [31].

2.1.5 Cognitive Modeling

Behavioral models are typically reactive in the sense that although the behavior exhibited by a behavioral model can be complex and natural, it is not the product of some greater plan but is instead solely determined by the model and its perception of the environment. Behavioral models have been used to simulate lower levels of behavior like flocking birds [59], swimming fish [72] and pets [11]. However, it is unclear whether behavior models will scale to the level of behavior required to simulate the intricacies and diversities of more complex behaviors such as those of humans.

In [28], Funge introduces the cognitive model as a means for controlling virtual characters in such a way that they appear to take deliberate, as opposed to reactive, actions. A main component of a cognitive model is *knowledge* about the world and how it can change. Another important aspect is *direction* or instructions to the character about how it should act to achieve certain goals. Funge uses a language based on first order logic to provide both knowledge and direction for virtual characters.

2.2 Architectures for Animation

Virtual characters are typically created for a particular purpose, whether it be to entertain us, teach us, interact with us, or represent us in a virtual world. The animation required for these applications can range in complexity from simple key-framed sequences for playback to artificial creatures that interact with other objects in their environment and may even exhibit some degree of autonomy.

As the complexity of applications increases, so does the complexity of the virtual characters used by the applications. Except for the simplest of characters, few animation models adequately meet a typical application's requirements either because the desired character behavior is not representable in a single model or the representation is too complex to implement efficiently. This is the case particularly for applications of autonomous creatures and artificial life. As a result, animation models are seldom used in isolation but are instead combined thus forming what we will refer to as an animation architecture. The following are some examples of projects and architectures that have been proposed for providing animation for virtual characters.

2.2.1 Subsumption Architecture

Although developed for robotics, the subsumption architecture proposed by Brooks [17] is considered one of the key architectural concepts for animating virtual characters. In this architecture, control is distributed across a collection of entities referred to as modules. Each module is a finite state machine with a set of inputs and outputs. The actions performed by the robot are not determined by a central controller but instead by the interactions between these modules. Some modules accept sensor input while others control the actuators of the robot. Connections between modules determine how sensor input translates into actuator output.

Modules are grouped together into behavior layers or levels of competence. Lower levels provide primitive behaviors such as collision avoidance while higher levels provide more sophisticated behaviors such as searching through an environment containing obstacles. Modules of higher levels can inhibit the outputs of or override (subsume) the inputs of lower level modules thus allowing higher level behaviors to influence lower level ones.

2.2.2 Artificial Fish

Tu and Terzopoulos [72, 71] have developed an architecture for artificial fish which is one of the few architectures that illustrates the combination of an underlying physics-based model with a behavioral model. A physics-based model is used to model the

fish and their underwater environment while a behavioral model is used to determine the most appropriate course of action given the current environment and internal state.

The behavior system consists of several components. The “intention generator” consists of a set of rules for determining which primitive behavior to engage in. The rules are predicates based on sensor input and internal state. Primitive behaviors include avoiding obstacles, schooling, escaping from predators, chasing, eating, mating, wandering and leaving. Once a primitive behavior is selected, the “focuser” extracts environmental information required by the behavior and the behavior uses this information to determine the appropriate motor commands to issue.

The motor commands make use of motor controllers, such as swim, turn left or turn right, that prescribe the muscle contractions that result in the desired motion. Each fish is modeled using a spring-damper model so muscle contractions are achieved by varying the rest length of the appropriate spring.

Funge [27] built his cognitive modelling system on top of this architecture. Using a language based on first-order logic, he built cognitive models that can plan out courses of action based on knowledge and the conditions in the surrounding environment.

2.2.3 Hamsterdam

In Hamsterdam, Blumberg [10, 12, 11] proposes a layered architecture that consists of a geometric model, a motor system and a behavior system. The motor system consists of a set of procedures, referred to as motor skills, which control the geometry through a degree of freedom layer which exposes the parameters of the geometric model to the motor system. The degree of freedom layer also ensures that higher priority motor skills have exclusive access to the parameters they control and that lower priority motor skills are only activated if the degrees of freedom they control are not affected by higher priority motor skills. The behavior system activates motor skills through the command layer which translates generic commands such as “move forward” into specific motor skills such as “walk” and also uses these commands to determine motor skill priority.

The focus of Blumberg's work is the behavior system which is designed using an ethological basis. Although not the first behavior system to incorporate concepts from ethology, the design of most aspects of the behaviors and their interactions was derived from ethology. Blumberg advocates the use of a "common currency" when providing information to the behavior model which amounts to using scalar values as inputs to behaviors. Additionally, sensor input is provided to behaviors through releasing mechanisms that translate sensor input into continuous values that are greater than or equal to zero. These values are combined with the values of internal scalar variables, representing creature state such as "hunger" or "fatigue", in order to determine the overall value of a behavior.

Competing behaviors are grouped together and their behavior values are used to determine the winning behavior. Associated with each behavior is either a command to be issued to the motor system or a sub-group of competing behaviors which is recursively evaluated to determine the winner at that level. The final winning behavior issues the primary motor command. Losers at each level can issue secondary commands, that will be executed if they don't interfere with the primary command, and make suggestions that may be used when activating a motor skill.

2.2.4 Improv

The Improv system [54, 55] was designed for behavior scripting that is suitable for directing virtual actors. The overall architecture is similar to that proposed by Blumberg but the behavior system is not based on a behavioral model but is instead a script that directs the actor.

The motor skills, or actions, are divided into groups of competing actions in which only one may be active at a given time. Action groups are further ordered such that more localized action groups, such as those that provide gestures, can override or subsume the effects of more global action groups, such as those dealing with locomotion. The system blends the effects of multiple actions during periods of transition between actions within a group or during action activation and deactivation to ensure that there is a smooth transition between actions.

2.2.5 Humanoid

The Humanoid project [13] is one of the many research projects related to virtual human animation at Miralab. The focus of the Humanoid project is the modeling of humanoids for real-time animation. Because of the behavioral complexity of humans, there is less of a focus on behavioral models and more of a focus on creating models that provide realistic motion that could be controlled by a higher-level behavioral system or by the user. For instance, some of the main areas of research include models for walking, grasping and facial animation.

An interesting architecture produced by this work is AgentLib [14]. This is a system designed to allow the blending of effects from motions produced by different types of motion generators, such as keyframe interpolation, inverse kinematics and motion capture. Atomic motions are encapsulated in an action which can be activated by a higher-level system. Also associated with the action are transition constraints for determining how an action is introduced and removed and priority modes for determining how the effects of multiple active actions are combined to control the joints of a geometric model.

2.2.6 Others

The Jack system [5, 6] was built for bio-mechanical modeling for human factors analysis. The low-level motions of Jack are governed by a constraint-based dynamic model and include collision avoidance and walking behaviors with balance control. Higher-level behavior control is provided by parallel transition networks (Pat-Net) [6] which are essentially a network of interconnected nodes implemented in a general purpose programming language.

The focus of Johnson's WavesWorld [36] was to develop an application and framework that eased the development of virtual characters. He provided tools for building geometric models and for controlling these models through a set of articulated variables. This framework combined motor skills and behaviors into a single layer that consists of sensor, skill and goal agents that interact to determine the values of articulated variables. It also included facilities to allow multiple agents to control a single

variable in such a way that their effects are blended.

Another interesting architecture is that of the popular Sims game [47]. Although there is no literature describing the details of the implementation, the authors have provided hints that behaviors are associated with distinct objects in the world rather than with the character itself. In this way, characters don't have to deal with all the possibilities but rather temporarily acquire abilities when they interact with specific objects. Thus, for example, a character that can only walk might suddenly acquire the ability to play pool when a pool table is introduced into its immediate vicinity. The ability to add behaviors in this manner to a character is made possible by limiting the types of characters that can inhabit the world.

2.2.7 Summary

Most of the architectures presented in this section represent significant contributions to the fields of computer graphics and animation. However, they are hard to compare as each focuses on a different aspect of interactive animation. For instance, the main contribution of Tu's work is the physics based model for simulated fish, while the main contribution of Blumberg's work is the comprehensive behavior hierarchy for providing autonomous behavior to a simulated dog. Each of these architectures required a large amount of effort to implement and was created specifically for the models being investigated. Therefore, even though it is conceptually possible that aspects of two or more of these architectures could be combined, to do so could potentially be almost as much implementation effort as the original architectures themselves, especially considering that the code for these architectures are not typically publicly available. As we will reveal, overcoming this shortcoming is one of the main goals of this thesis.

2.3 Abstractions for Animation

In order to promote the reuse within animation models, we must first identify common abstractions that are used in implementing them. This section presents generalizations of several of the abstractions that are commonly used when creating and representing virtual characters and their animation models.

2.3.1 Levels of Abstraction

Architectures typically have a high level abstraction that represents an individual character. Depending on the application, a character may be referred to as an object, agent, actor, or creature. Architectures typically represent aspects of a character at different levels of abstraction and combine or layer these levels of abstraction together in order to create a complete character model. The models presented in Section 2.1 play a large role in determining these levels of abstraction. For instance, lower-level abstractions, such as motor skills, are often represented using a kinematic model while higher level abstractions, such as decision making or goal determination, are more frequently represented using a behavioral model.

Structure

The base level of abstraction for virtual characters is the geometric model. This abstraction layer can be thought of as forming the structure of the character as it represents the matter that constitutes the character. There are many commercially available platforms for representing and displaying geometric models like the ones described in Section 2.1.1.

Physics-based models are a higher level abstraction that control a geometric model. However, we consider models of this type to also be categorized as a structural abstraction because these models are tightly coupled to the underlying geometric model. In other words, physics-based models typically require exclusive control of all geometric parameters that they control to ensure continuity and stability of the model's system of equations.

Articulated Variables

The structure of a character is controlled either through a set of parameters representing point locations, joint angles or other attributes of the geometric model or through a set of parameters that influence the forces and torques present in a physics-based model. These control parameters have also been referred to as degrees of freedom, articulated variables, effectors or actuators and a character is animated by changing the value of these parameters over time.

In most cases, this level of abstraction is used to form an abstraction barrier between the structural abstraction and higher level abstractions in order to allow the implementation of higher level abstractions to be somewhat isolated, and independent of, the implementation of the structure. Thus, higher level abstractions are simply concerned with providing values for the articulated variables of a character. Facilities may be included to ensure that exclusive access is granted to a particular variable or that multiple contributions to the same variable are combined or blended together appropriately.

Actions and Controllers

Varying the value of articulated variables over time animates a character. Varying the values of several variables over time in a coordinated fashion can produce some sort of action whose purpose is either to achieve some goal or convey information, such as the emotional state of the character. Several terms have been used to describe these types of coordinated motions including actions, motor skills or motor commands. In essence, each action is a procedure that influences one or more parameters in order to produce coordinated motion. As with articulated variables, an action abstraction forms a further abstraction barrier between higher level abstractions and the structure of a character. This frees higher level abstractions from dealing with individual articulated variables and allows them to simply identify the actions that should be active at a given time.

Associated with an action is one or more controllers which act to determine the specifics of the action as time progresses. Controllers map environmental conditions

and character state into output that is incorporated into the effectors associated with the action. As a result, higher-level entities can control actions by providing the appropriate input to the action's controller. Thus controllers form an abstraction barrier between actions and the structure of the character and its environment and between higher-level abstractions and actions.

Behaviors

The purpose of the behavior system is to determine which action is most appropriate given the current circumstances. The behaviors of a behavior system process a character's sensor input and internal state and interact with other behaviors in order to determine the most appropriate course of action. Thus, this level of abstraction corresponds to behavioral models.

The differences between a behavior and an action can be a bit fuzzy. Indeed, some architectures do not distinguish between the two. However, in most architectures, there is a distinct difference between what we refer to as actions and behaviors. In [74], Tyrell defines an action as the most primitive unit of behavior. Therefore, from a behavior standpoint, actions form an abstraction barrier between the behavior system and lower level abstractions such as a creature's structure. Tu [71] claims that the separation of action selection from motor control is necessary to achieving high level behaviors in most creatures, especially complex ones.

2.3.2 Organizational Abstractions

In many cases, animation models are built by combining multiple components that may be instances of one or more of the abstractions presented in the previous section. As the number of components that constitute a model increases, so does the complexity of the model. The effort of creating and modifying these models is reduced by arranging these components using various organizational abstractions. The specific arrangements chosen determine how groups of these components interact. Different organizations are used to serve different purposes and are often combined to further reduce the complexity of the resulting model.

Group

The simplest organizational abstraction is a group of components, a generalization of the other organizational abstractions presented below. However, we consider a group to be a set of components that are either independent of each other or interrelated in a common way.

For instance, an action of the Improv system [54] is a group of functions that provide values for an appropriate set of degrees of freedom. The relationship between these functions is the time input used to determine the function output. The Improv system also groups independent actions together in competing action groups such that only one action in the group can be active at a particular time. In a similar way, Blumberg uses behavior groups to provide mutual exclusion between a set of competing behaviors. The behaviors of the action-selection mechanism used by Tyrell [74] are grouped together in what he refers to as a system. The behaviors of a system are devoted to the control of behavior related to a particular sub-problem such as getting food, escaping from predators, or sleeping.

Hierarchy

The components that make up a group can be arranged in a hierarchy which implies some sort of ordering of the members in the group. For instance, the action groups of the Improv system are further organized in an ordering where the effects of action groups containing more global actions can be overridden or modified by the action groups of more localized actions. Another example of a hierarchy is found in Hams-terdam where a behavior can be the parent of a child behavior group that is evaluated when the parent behavior is the winning behavior in its group. Hierarchies are used extensively in behavior systems including those of Tu [71] and Tyrell [74]

Network

A network is a group of components that are interconnected in such a way that the output of one component in the network acts as the input to one or more other components. Traditional neural networks [50] consist of connected nodes where each

node has multiple inputs and a single output that is a function of those inputs. Brooks' [17] subsumption architecture consists of a set of interconnected modules where the outputs of the modules are connected to the inputs of other modules. In Johnson's WavesWorld [36], a network of sensor, goal and skill agents forms the animation model of a character.

2.3.3 Meta-Level Abstractions

Components that match the abstractions presented in Section 2.3.1 may be arranged using one or more of the organizational abstractions presented in Section 2.3.2. The exact nature of the interactions between these components is defined by relationships that we refer to as meta-level abstractions.

We have borrowed the term "meta-level" as it applies to programming languages. A meta-level facility is a feature or functionality that is provided by or defined by the underlying semantics of a language. For instance, message passing or class inheritance are meta-level facilities of an object-oriented programming language. Most languages do not allow a designer to change how these meta-level facilities work. However, some languages, such as Smalltalk and Java, do support reflection which is basically the ability to modify meta-level facilities.

From the stand-point of character animation, meta-level abstractions define how the entities within an architecture layer interact as well as how interaction between layers is handled. In most architectures, these abstractions are built into the underlying architecture and their semantics is not modifiable by the animator.

Information

An important aspect of any animation model is the information that makes up the state of the model or the entities that constitute it. As one would expect, the basic abstraction used to represent information within a model is the variable.

The traditional use of variables involves the assignment of a value to a variable with subsequent assignments overriding previous ones. References to a variable result in the most recently assigned value. For instance, in Brooks' [17] subsumption

architecture, module variables are assigned values when the module enters a side effect state associated with that variable. In Blumberg's [11] architecture, the primary motor skill assigns values to the effected degrees of freedom on each animation tick. The values associated with these types of variables may be simple or complex data structures.

However, it is common for the value of a variable in a behavioral model to be determined by a function or calculation that is re-evaluated each tick. These types of articulated variables typically represent scalar values or other simple data types. The behavior values and internal variables of Blumberg's behavioral model are represented using these types of variables.

Communication

Communication is a transfer of information between components. Within models for computer animation [36, 11, 71], communication between components is often accomplished using shared variables. For robotics [17], connections are used to transfer information from the output of one component to the input of another. Communication between architectural layers is also often achieved using virtual connections.

Hand in hand with communication is the concept of an interface. An interface determines how elements within an abstraction level communicate and interact with elements outside the abstraction and hence act as an abstraction barrier that isolates one abstraction from another thus simplifying model creation.

Combination

The value of a particular variable in a model is often a function of the state of the model. In other words, the variable's value may be a function of one or more of the model's variables. The function associated with a variable is typically determined by the components of the model and their relationships. For instance, in AgentLib [14], the value of a particular joint angle is determined using the weighted combination of the difference between the desired angle for each active action that affects the joint and the previous joint angle. This sum is then added to the previous joint angle to

determine the new angle $(\theta(t) = \theta(t - 1) + \sum_i (\theta_i(t) - \theta(t - 1)) \cdot \omega_i(t))$.

We refer to these types of functions as either combination schemes or blending schemes since they are used to blend or combine multiple values or contributions into a single value. There are several different schemes that have been widely used in animation architectures. The simplest blending scheme is the weighted combination of contributions as in the AgentLib example mentioned above. The weights are typically between 0 and 1 and in many cases the weights are normalized such that the sum of the weights is between 0 and 1.

The effects of other components can also be represented using gains which are similar to weights but not subject to the same restrictions. This approach is used by Blumberg. The value of a behavior controls an internal variable via a gain, either positive or negative. Also, the behaviors within a group each use an inhibitory gain to other behaviors in the group which ensures that there will be a winning behavior [11].

Primitive

A primitive is a component at the lowest level of abstraction within an architecture. In essence, primitives are the building blocks from which the models of an architecture are constructed. For instance, an architecture will have primitives for connecting sensor input to an animation model or for forwarding model evaluation results to the parameters or effectors of a structural model. Certain primitives, such as those that provide information storage, communication and combination are not readily tailorable and thus dictate certain aspects of the models built using the architecture. This leads to specific abstractions being used within an architecture that do not readily support reuse and sharing with other architectures. Architectures may allow additional primitives to be implemented in a general purpose programming language and added to the architecture.

2.4 Patterns

A pattern is simply a recurring temporal and spatial relationships between entities. Of particular interest to us is the behavior patterns exhibited by animals since many behavioral models are derived from the study of animal behaviors [11, 71, 74]. The architectures supporting these models often include primitive abstractions derived from the study of natural behavior.

2.4.1 Model Representation

It is our interest to define a framework that promotes reuse within animation models. Therefore, we need to identify a suitably low level of abstraction that is capable of representing the different higher-level abstractions mentioned in Section 2.3. Synergetics and Dynamic Pattern Theory provide a low-level view of behavior patterns that has inspired our representation.

Synergetics

Synergy is often taken to simply mean working together. A synergetic system is, then, a system of parts that work together in a coordinated fashion. However, in [26], synergy is defined as “the behavior of whole systems unpredicted from the behavior of their parts taken separately”. Of course, the parts or components of a system can often be viewed as systems themselves which can be further decomposed into sub-components and so on. This leads to a hierarchy of components where the behavior of a component is not predictable from the behaviors of its sub-components. So, in [26], synergetics is described as “the exploratory strategy of starting with the whole and the known behavior of some of its parts and the progressive discovery of the integral unknowns...”.

Haken [34] presents a more formal and mathematically rigorous experimental approach to synergetics in which the spontaneous formation of spatial, temporal or functional structures or patterns of behavior in physical, chemical and biochemical systems is observed and characterized. Haken’s Synergetics provides a general approach to characterizing self-organizing systems and the conditions that bring about

this self-organization.

Self-organization refers to the phenomenon observed in many systems that involves very high dimensional subsystems (i.e. many degrees of freedom) acting in a coordinated fashion that make the system appear to be low dimensional (i.e. few degrees of freedom). There is no controlling influence, either external or internal to the system, that determines the structure or pattern that the system exhibits. Instead, the components of the system arrange themselves into an organized pattern in response to conditions in the surrounding environment.

Although, microscopically, a self-organizing system may exhibit a seemingly large number of degrees of freedom, a small number of degrees of freedom may be sufficient to describe the macroscopic properties of the system. This is referred to as the *slaving principle*. The describing degrees of freedom are referred to as *collective variables* or *order parameters*, θ , while the affected variables are referred to as *enslaved variables*, e . Changes in the organization of a system are brought about by changes in *control parameters*, α . As the control parameters change, the collective variable dynamics may change as well. This, in turn may cause new patterns to arise or dissipate spontaneously. The control parameters may be unspecific to the resulting patterns or, in other words, the control parameters may not play a role in defining the pattern's behavior. Instead, control parameters lead the system from one pattern to another.

Of interest to us is the relationship between the collective variables and the control parameters. More specifically, in Synergetics, the dynamics of a collective variables, θ_i are described as a function of the collective variables and the control parameters.

$$\dot{\theta}_i(t) = f(\theta, \alpha) \quad (2.1)$$

In a similar way, we model behavior as a collection of articulated variables whose dynamics are determined by a function of the variables in the system.

Dynamic Pattern Theory

Dynamic pattern theory was proposed by Kelso [62, 39, 38]. The main focus of dynamic pattern research is to describe observed spatial and temporal behavioral

patterns not in terms of the effects of the patterns, but instead, in terms of the dynamics of pattern formation and, hence, the transitions between patterns. Dynamic pattern theory relies on synergetics [34] to describe behavior patterns in terms of their dynamics.

One experiment, whose results were adequately described using synergetics, involved the oscillation of two index fingers at the same frequency. The frequency of oscillations was controlled by a metronome which acted as a control parameter for the experiment. The collective variable or order parameter observed was the phase relation between the fingers. The phase relationship was found to be more stable for certain values. For lower frequencies, the in-phase and anti-phase relationship between the fingers was found to be the most stable. As the frequency increased, the anti-phase pattern becomes unstable and a transition to the more stable in-phase pattern eventually occurred.

An important aspect of this experiment is that the subjects tested were instructed not to intentionally try to prevent a switch in the pattern if they should detect that the phase relationship between the fingers was changing. Thus, the experiment tested the coordination between the index fingers independently of the subject's intentions. However, in biological systems, especially in humans, intention plays a role in determining what patterns a system may exhibit. Dynamic Pattern Theory is concerned with applying the descriptive techniques of Synergetics to patterns observed in biological systems. However, it also attempts to incorporate the concepts of environment, memory and intention into the equations that describe the collective variable dynamics of self-organizing systems [62].

$$\dot{\theta}_i(t) = f(intention) + g(memory) + h(environment) \quad (2.2)$$

In order to describe the relationship between dynamic patterns, Kelso [38] describes a *tripartite scheme*. His claim is that understanding at any level of organization starts with the knowledge of three things: the boundary conditions (including the control parameters), the interacting elements or components (including enslaved variables) and the emerging patterns (characterized by the collective variables and

their dynamics). He further claims that this scheme can be used to describe many different levels of organization in a whole system and that the variables in a system play different roles at different levels of organization. For example, the boundary conditions at one level may be the collective variables or enslaved variables of another or vice versa. Thus, the levels can be viewed independently but form a hierarchy that describes the complete system.

Of particular interest to us is the following relationships between variables that are implied by this theory.

- A system may be divided into levels of organization and may therefore be decomposed into sub-systems thus forming a system hierarchy. We make use of such a hierarchical arrangement for organizing the variables within our model representation.
- The functions that describe the dynamics of a variable are the combination of contributions from different sources, be they memory, intention or environment. Therefore, we model the functions that determine the dynamics of the variables of our representation as a combination of contributions from multiple sources.
- The variables within a system may play different roles at different levels of organization. Therefore, we do not enforce any restrictions on the interconnection structure between the variables within a system or any of its subsystems.

2.4.2 Model Specification

We propose that a representation based on the concepts presented in the previous section is capable of representing many of the abstractions presented in Section 2.3. However, the complexity of directly specifying anything but the simplest of animation models using such a representation would be too difficult and tedious to be of any practical value. Therefore, in order to reduce the complexity of model creation for this representation, we require a mechanism that supports the definition and use of higher-level abstractions. Alexander's work on Pattern Languages of Architectural has inspired our approach to defining and using the required higher level abstractions.

Patterns for Architecture

Christopher Alexander [2] proposed a pattern language for building and planning that could be used by both professionals and novices to create designs and communicate design ideas. The goal of his pattern language is not just to provide instructions for designing buildings and communities but to help create designs which contain what Alexander calls the “quality without a name” or the quality of “life”. In [1], Alexander states that “every place is given its character by certain patterns of events that keep on happening there”. Furthermore, these “patterns of events are always interlocked with certain geometric patterns in space”. Certain patterns enhance the life of the inhabitants while others will, as Alexander states, “keep us locked in inner conflict”.

In his pattern language, Alexander strives to describe the geometric patterns that appear in communities, buildings, and rooms that bring about the above mentioned quality. This is often not a straightforward process as it not only involves observing many different existing structures that have this quality but it also involves recognizing what relationships between the components present are essential to the existence of the quality. It is these relationships that are the patterns of interest. However, Alexander’s aim was not just to identify the patterns but also to identify when these patterns should be present and to describe how to apply them to particular situations. Furthermore, the goal was to make this pattern language accessible to both professionals and novices. For this reason, why a pattern works in a certain situation is also considered part of the pattern description.

To this end, Alexander presented a pattern description, or, what he referred to simply as a pattern, as a general solution to a recurring problem. Each occurrence of the problem has its own unique characteristics. Consequently, the solution to the problem is stated in such a way as to allow many different variations of the solution to be formulated depending on the forces involved. Since each occurrence of a repeating problem is unique, each solution is unique as well.

The patterns that occur in architecture are not isolated. Often, a pattern is recognizable by the smaller scale patterns that help make up the pattern. Also, the larger scale patterns that exist in the surrounding environment play a role in

defining the pattern. Consequently, the patterns in Alexander's pattern language are arranged in a hierarchy where the patterns near the top level of the hierarchy deal with large scale design issues such as the layout of regions and towns while the lower level patterns deal with structures such as buildings or parks. The lowest level patterns deal with details such as room layout and materials. Some patterns deal with problems that are structurally significant while others deal with relationships that, although perhaps not structural, help the design "live".

Each pattern is divided into three parts. The **context** of the pattern describes *when* the problem occurs. The context identifies the conditions in the surrounding environment that bring about the problem including any larger-scale patterns that this pattern helps to complete. The **system of forces** describes the conflicting forces that influence the pattern and describes *why* the pattern works in satisfying these forces. The **configurations** describe *how* to resolve the conflicts between forces including the smaller scale patterns that are needed to complete, to fill out and to embellish the pattern.

Instead of using patterns to describe the relationship between architectural entities such as rooms, doors and windows, we propose to use patterns to specify the architecture of an animation model. Therefore, instead of being concerned with building materials, we are concerned with the materials used to build animation models. For our work, these are the variables that constitute a behavior system and the relationships between them. Also, instead of being concerned with describing the patterns in terms understandable by human architects, we are concerned with defining patterns in a manner that can be used by a compiler to build animation models.

The following features are a summary of the aspects of patterns that are important in our work.

- A pattern defines a relationship between a set of entities and smaller-scale patterns.
- Different entities or arrangements of entities that result from the application or realization of a particular pattern are not identical. A pattern is a general specification of a relationship that is made specific by the particular entities

that are configured using the pattern and any smaller-scale patterns that are involved in the realization.

- The exact entities and smaller-scale patterns that are used to realize a pattern are determined by the context in which a pattern is applied.
- The context of a pattern is provided by larger-scale patterns.

We define abstractions using patterns. For instance, a virtual character can be viewed as a pattern that defines a relationship between a structural or geometric model, a set of actions, and an action-selection mechanism. An action can be viewed as a pattern that defines a relationship between the name identifying the action (e.g. “Walk”), a coordinating time factor and a set of effects on the variables representing the joint angles or other control parameters of a character. A larger scale pattern, such as an action group, may provide a context that provides the time factor used to coordinate the actions in the group while smaller-scale patterns may be used by an action to determine how its effects are combined with other actions that affect the same parameters. At the top of the pattern hierarchy are patterns representing the whole character or an animation model of a character. These patterns provide the context for the various sub-models, such as an action-selection model or an action-combination model, that make up the character’s animation model while the smallest-scale patterns determine how the actual variables in the system representing the animation model are affected by the value of other variables in the system.

Chapter 3

Virtual Behavior Model

In this chapter, we describe the computational model we use to represent animation models for virtual characters. We are not concerned with reuse at the representation level but instead our aim is to promote the reuse of animation model elements at the specification level. Therefore, the goal in this chapter is to define a representation whose focus is not the building of animation models but is instead the execution of animation models for the purpose of controlling virtual characters.

In this chapter, we start by presenting the requirements of an animation model representation. We then present the Virtual Behavior (ViB) model that was inspired by Synergetics [34] and Dynamic Pattern Theory [38] as a means of meeting these requirements. Since the goal of the representation is to provide behavior for virtual characters, an important aspect of this representation is the implementation of a supporting architecture and evaluation engine. We will then discuss the data types and objects used to constitute ViB models followed by a description of the important aspects of our implementation of the ViB engine which evaluates ViB models for the purpose of animating virtual characters.

3.1 Requirements

There are several requirements that must be met in order to adequately represent animation model elements for execution purposes while also promoting reuse at the

specification level.

3.1.1 Simple

The representation must be as simple as possible in order to ease implementation of the supporting architecture and compilation of model specifications into the representation. This is required in order to increase the portability of the model specifications. Just as object-oriented languages such as Smalltalk and Java have virtual machines that allow the same code to be used on different platforms, we desire a representation that can be used by different architectures to provide character animation.

3.1.2 Flexible

The representation must be flexible in the sense that its use should not be restricted to representing a single animation model type but instead should be able to represent different types of models such as those incorporating kinematic and behavioral aspects as well as physical notions. Furthermore, although the representation need not provide facilities for directly defining higher-level abstractions, it should be capable of supporting the different abstractions mentioned in Section 2.3 such as actions, behaviors and action-selection and meta-abstractions such as blending schemes.

This type of flexibility allows represented models to be potentially used as a complete animation model for virtual characters, providing the actions and behaviors that determine the character motions, or as partial models that provide only a subset of these. Therefore, the supporting architecture must provide an interface to lower-level architectures that may provide geometric, physics-based or kinematic models and to higher-level architectures that provide behavioral or cognitive control.

3.1.3 Natural

The representation must also be capable of representing natural behaviors. In other words, we want to be able to build behaviors and behavior systems that are based on observations of natural systems such as animals. However, we do not want to restrict

ourselves to animal behaviors or to a particular area of behavior study. Therefore, we desire a representation with a suitably low-level of abstraction that is capable of representing higher level behavior abstractions.

3.2 The ViB Model

The basis of our proposed model is inspired by the inter-disciplinary fields of Synergetics and Dynamic Pattern Theory which are concerned with describing behavior patterns in natural systems. We have adopted some of the concepts from these fields and applied them to a fixed sampling rate or time-step evaluation in order to prescribe behaviors for virtual characters. This provides us with a simple, low-level model for animation that is capable of representing naturally occurring behaviors. We refer to the resulting representation as the Virtual Behavior (ViB) model. A ViB model consists of a set of time-varying, interconnected channels which define the state of the model, and which are organized in a system hierarchy.

3.2.1 Channels

In Synergetics, the state of a system is described by a set of continuous variables. In a ViB model, a *channel* corresponds to a system variable and, therefore, the state of a model is provided by a set of channels,

$$\mathbf{c} = (c_1, c_2, \dots, c_M)$$

where $\mathbf{c}$ is the set of all channels, c_i .

A model for rigid body motion could contain channels representing the location or orientation of a body in its environment.

$$\mathbf{c} = (\text{location}, \text{orientation})$$

A model for articulated figures could contain channels representing the angles of each of the joints of the figure while a model that provides action selection could contain

channels representing the activation level of the actions of the model. Higher level models could contain channels representing internal state variables such as hunger or fatigue and may also contain channels representing sensor input or memory.

3.2.2 Channel Output

The state of a model at time t is defined by the *output* of the channels of the model at that time.

$$\mathbf{c}_t = \bigcup_{i=1}^M c_{it}$$

where $\mathbf{c}_t$ is that state of the model at time t and c_{it} is the output of channel c_i at that time.

3.2.3 Channel Signal

The output of a channel is determined from a *signal* that is attached to the channel. The result of evaluating the signal at time t is used to *delta* the output of the channel on the previous iteration, $t - 1$, in order to determine the current channel output.

$$c_{it} = c_{i(t-1)} + s_i(\mathbf{c})$$

where $c_{i(t-1)}$ and c_{it} represent the output of channel i at time $t - 1$ and t respectively and s_i is the signal attached to channel i and is a function of the state of the model.

This approach is derived from Synergetics which describes the collective variables of a system using the dynamics, or derivative of the variable's behavior. However, for efficiency reasons, the signal of a channel outputs a delta instead of a derivative. A similar approach has been used often in other work [14]. However, the significance here is that it provides a unified approach to channel output determination which simplifies the implementation of the supporting architecture as well as the blending of multiple contributions as described in Section 3.2.4. The approach is made more

flexible by allowing the delta operator to be a tailored or changed for a particular channel depending on the type of output needed or the purpose of the channel. For instance, the output of a channel signal can be used as the channel output directly by using the REPLACE delta operator that simply uses the signal output as the channel output while ignoring the output of the channel on the previous iteration.

When describing channels and signals, we use a notation that places the delta operator above the assignment symbol. Therefore,

$$c_{it} \overset{+}{=} s_i(\mathbf{c})$$

is equivalent to $c_{it} = c_{i(t-1)} + s_i(\mathbf{c})$. For example, in a behavioral model, the hunger level of a character may increase steadily over time. The following hunger channel would increase the hunger level of the character by 0.1 per second.

$$hunger_t \overset{+}{=} (0.1\Delta t)$$

where 0 is the lower bound on $hunger_t$ and the upper bound depends on behaviors that reduce hunger, an example of which is presented in the next section.

3.2.4 Channel Contributions

Dynamic Pattern Theory states that the function that defines the dynamics of a collective variable is the *blend* of contributions from multiple sources. In a similar manner, the signal of a channel is the blended combination of the output of multiple functions.

$$s_i(\mathbf{c}) = \sum_j f_j(\mathbf{c})$$

where f_j represents the contribution function from source j to channel i and may be a functional combination of the output of one or more channels in $\mathbf{c}$.

For example, the location of a body at time t depends on its previous location, its velocity and any acceleration that results from forces acting on the body. Therefore, a suitable function for determining the location is

$$location_t \pm \Delta location_{t-1} + \sum_i a_i(t) \frac{\Delta t^2}{2}$$

where $\Delta location_{t-1}$ is the previous change in the location and represents the effects of inertia given a constant Δt and a_i is the acceleration resulting from force i . Similarly, the hunger level of a creature may increase steadily over time but will also be reduced while the creature is eating.

$$hunger_t \pm (0.1\Delta t) + (-1 \times eating.activation_{t-1})$$

A contribution to a channel will typically be weighted in some way, whether it is done independently of or dependently with other contributions to the channel. However, the ViB model does not provide built-in weighting schemes. It only provides facilities for representing functions as described in Section 3.3.4. This places less restrictions on the types of weighted combination that can be used in a model. Therefore, the ViB model only provides low-level blending support in the sense that the output of the contribution functions is combined using a binary operator that is both commutative and associative so that blending is order independent. Like the delta operator, the default blend operator used to combine multiple contributions is addition but the blend operator of a channel may be changed as required.

3.2.5 Systems

Dynamic Pattern Theory states that there are different levels of organization within a complete system. For instance, within a biological organism there is physical and chemical level behavior patterns at the microscopic level. At the macroscopic level, an organism may exhibit behavior patterns that convey intelligence. To reflect this style of organization, the channels of a ViB model are contained in a *system*. Within

a system, channels can be further sub-divided between *sub-systems* thus forming a system hierarchy.

For example, the system associated with a particular creature could contain the complete animation model used to control the creature while the sub-systems may represent atomic actions or behaviors, sub-components, sensed objects, or other collections of data representing the abstractions used to generate the representation. This is illustrated in Figure 3.1 which shows a system representing a creature (Bob) with sub-systems for components (leg and arm), behaviors (find food) and actions (walking). More specifically the ViB model for *Bob* may contain channels representing joint angles (*Right.Elbow.flexion*), action activation (*Walk.activation*) or sensor input (*SensedObjects.foodSource*). The output of the joint angle channels is influenced by the relationships between other subsystems in the model. For instance, the hunger value in combination with the presence of a sensed food source may affect the FindFood value which is turn may activate the Walk action resulting in changes to the joint angles.

Figure 3.1: Example System of Channels

3.3 ViB Model Entities

In our implementation of the ViB architecture, the entities that make up the ViB model are represented as first class objects. In this section we summarize the objects used to constitute a ViB model.

3.3.1 Channel Output

The output of each channel is a particular type of data. For instance, a channel representing a *location* as mentioned in Section 3.2.1 would output a point representing a position in the environment while the *orientation* channel would output a value representing a rotational transformation. An action *activation* level could be a real number between 0 and 1 or a boolean indicating whether the action is active or not. A sensed *foodSource* could be another ViB model whose channels contain information about the food source.

Since the purpose of the ViB model is to animate virtual characters, the data types supported by the architecture correspond to those commonly used in computer animation. The core set of data types are:

- Integers¹
- real numbers represented using double precision Floats
- 2D and 3D vectors of real numbers (Point and Point3D)
- rotations represented using Quaternions
- Booleans.

Suitable default delta and blending operators have been identified for each of the supported data types. These operators are described in Section 3.3.3.

In addition to the above data types, a channel can also output a ViB model object. This allows information about sensed objects or other objects to be incorporated into a model as a single unit instead of creating separate channels for each relevant attribute

¹We capitalize class names mentioned in the text in order to differentiate classes defined in the implementation from the concepts being discussed.

of the sensed entity. There are special functions provided by the architecture to access the output of individual channels within such a model. These are described in Section 3.3.4

3.3.2 System and Channel

The systems and channels of the ViB model combine to form the hierarchical organization of the model. The System and Channel classes correspond directly to these ViB model concepts. A System consists of a set of channels and systems while a channel contains a signal as described in Section 3.3.3.

System and Channel Naming

All systems and channels are named. This allows channels to be identified and referred to for the purpose of referencing their output. A channel is identified by the ordered set of names starting at the root of the system hierarchy and following the system branches to the system containing the channel in question. For instance, the channel identifier *Bob.Walk.activation* refers to the *activation* channel that is contained in the *Walk* sub-system of the *Bob* system as shown in Figure 3.1.

The names of the channels directly contained in a system must be unique as must the names of any sub-systems within a system. However, a system may contain a channel and a subsystem of the same name. This allows a channel to have a corresponding system of the same name that contains channels used specifically by the contribution functions associated with the channel. For example, if the change in the angle of flexion of a joint is determined by the normalized weighted sum of a set of contributions, then a possible channel signal for determining the joint angle would be

$$flexion_t \pm \sum_i (\Delta\theta_i \cdot \frac{\omega_i}{\sum_j \omega_j})$$

where $\Delta\theta_i$ and ω_i are the contribution identifying the desired change in angle and the weight associated with this contribution from source i respectively and $\sum_j \omega_j$ is the

sum of all contribution weights to the channel which is used to normalize each ω_i .

A more efficient implementation accumulates the total weight of all contributions to the channel in another channel and uses the output of this second channel to normalize the weights. Since this *totalWeight* channel is associated with the *flexion* channel, it is made a child of the system with the same name.

$$\begin{aligned} flexion_t &\triangleq \sum_i (\Delta\theta_i \cdot \frac{\omega_i}{flexion.totalWeight_t}) \\ flexion.totalWeight_t &= \sum_i \omega_i \end{aligned}$$

Channel Accessibility

In behaviors described by Dynamic Pattern Theory [38], the variables that describe the state of the system can play different roles at different levels of organization. As an example, consider a variable that describes the degree of hunger of an organism. At the chemical level, the degree of hunger may be an enslaved variable meaning that the degree of hunger is solely a function of the chemical processes involved. However, at a higher level, hunger may be a collective variable of the resulting hunger pangs if the hunger level directly describes the magnitude or frequency of the pains. At an even higher level, hunger can act as a control variable for the organism's decision mechanism in the sense that different degrees of hunger may trigger different behaviors but the degree of hunger does not reflect anything about the state of the triggered behaviors.

Consequently, the system organization of a ViB model does not impose any restrictions on the interactions between channels. In other words, a contribution to a channel may be a function of any of the channels in the ViB model no matter what the relationship between their containing systems. Therefore, from the standpoint of the ViB engine, the hierarchical arrangement of sub-systems and the channels they contain is mostly arbitrary and should represent a logical organization of channels based on the abstractions used to build them. The only effect the system organization has on a ViB model is in the determination of the channel evaluation ordering

that results from signals that are a function of the current output of other channels in the model. This is discussed in more detail in Section 3.4.2.

3.3.3 Signal

Each Channel has an associated Signal that is used to determine the output of the channel. A Signal consists of a set of contribution functions and is evaluated each time-step in the following manner.

1. Each of the contribution functions that constitute the signal are evaluated.
2. The output of each function is blended together using the channel's blend operator to form the signal output.
3. The signal output is applied using the channel's delta operator to the previous channel output to form the current channel output.

Therefore, each of the contribution functions that constitute a signal are independent except for the fact that they may have one or more inputs in common. Furthermore, the blend and delta operators play a large role in determining the channel output. Another important aspect of a signal is that it can be assigned an initial output that acts as the output of the channel at time $t = 0$.

Blend and Delta Operators

The general equation for channel output determination is

$$c_t = c_{t-1} \oplus \biguplus_i f_i(t)$$

where f_i represents the contribution function to channel c from source i , $\oplus$ represents the binary delta operator and $\biguplus$ represents the binary blend operator such that $\biguplus_i f_i(t) = f_1(t) \uplus f_2(t) \uplus \dots \uplus f_n(t)$.

Since there is no order associated with the contributions that are made to a channel, the blend operator should be order independent; i.e. $a \uplus b = b \uplus a$ (commutative). Generally, the delta operator should relate to the blend operator such that $c \oplus (a \uplus b) = (c \oplus a) \oplus b$. As one would expect, the contribution functions attached to a channel must output the same type of data object. Like the mappings defined by other functions, as described in Section 3.3.4, the default mappings for the $\oplus$ and $\uplus$ operators depend on the data type.

Scalars and Vectors: For scalar and vector types, the addition (+) operator meets the requirements mentioned above and provides the blending and delta behavior that is suitable for most situations involving these types.

Orientations: For orientations, which are represented in the ViB engine using quaternions, the blend and delta operator is multiplication. Multiplying two quaternions $q = q_1 \times q_2$ results in a third quaternion q that represents a rotation that is equivalent to applying q_1 as a rotation followed by q_2 [63]. Therefore, the multiply operator on quaternions is not truly commutative but does approach this characteristic for suitably small time intervals. The same can be said about using multiplication as the quaternion delta operator.

It should be noted that the delta applied to the previous orientation results in a rotation that is relative to the local frame of the previous orientation. Because the blend and delta operators do not truly meet the above mentioned conditions, this can result in an accumulation of errors caused by the ordering. Therefore, while quaternions may be suitable for free flying bodies, they may cause problems for bodies whose motion is relative to other bodies. Hence, like many other architectures, relative orientations, such as the joints of an articulated figure, are controlled by one or more scalar angles associated with fixed axes of rotation where the rotations are performed in a fixed order.

Booleans: By default, boolean values are blending using the OR operator and the results of the blend are used as the channel output, thus replacing the previous channel

output.

ViB Models: The ViB model delta operator always replaces the previous channel output. Blending is more complicated and involves the use of a special blending operator that provides comparison between channels within the ViB models to select one of the models being blended.

Tailoring Operators

In some cases, the use of other blend and delta operators is required to achieve the desired channel output. Therefore, these operators can be changed for an individual channel. The currently supported blend operators are

- addition
- multiplication
- minimum and maximum for scalar values
- AND and OR for boolean values.

The supported delta operators are the same as those listed above with an additional REPLACE operator that results in the signal output being used as the current channel output directly while ignoring the previous channel output. For instance, if a channel is being used to output the sum of a number of other channels, it is desirable to have the output of the signal replace the previous channel output. It is possible to accumulate the total without changing the delta operator by accumulating the change in all the values being summed but this may be subject to numerical computation error and is inefficient.

Other Signal Characteristics

A signal can be assigned an initial output which acts as the channel output for time $t = 0$. If unspecified, the initial output of a channel is $\emptyset$ which is a special system variable that represents the additive identity of all the data types supported by the ViB engine. Also, in some cases, the output of a channel should be restricted to

a particular range. Therefore, it is possible to specify a maximum and minimum value that a channel can output. Any channel output that falls outside this range is adjusted to fall on the appropriate boundary value.

3.3.4 Function

At the ViB model level of representation, there are no explicit facilities for defining component interactions such as action-selection or action blending. Therefore, it is the functional relationships between channels that are used to determine the behavior that results from the evaluation of a ViB model. These relationships are captured in the channel signals and their contribution functions.

The contributions made to a channel are represented as Functions. Like Signals, Functions, when evaluated, output a value of one of the supported data types. Although Functions may be simple, they are more often complex combinations of the output of one or more channels. A wide range of low-level functions are supported by the architecture and these functions can be combined in a function composition hierarchy in order to produce the complex functions required to control interactions within a model. Both general arithmetic functions and special functions related to computer animation are provided

Simple Functions

The simplest functions form the leaves in a function composition hierarchy and include

- constant functions that output one of the supported data types
- the current system time t
- special constants such as $\emptyset$ and π
- references to the output of other channels in the model
- references to models and the output of channels within those models.

Channel Output Referencing

Contribution functions are often functions of the current state of the model which is defined by the output of the channels that make up the model. Access to the output of a channel is provided using a special function referred to as a channel reference. Channel references normally represent the output of a channel for the previous iteration ($t - 1$) but can also be used to reference the output of another channel for the current time-step t . The recent history of a channel's output ($t - 2, t - 3, \dots$) is also available.

To illustrate channel output referencing, consider the following formula which can be used to combine the effects of several sources on a joint angle [14].

$$\theta_t \triangleq \sum_i (\theta_i(t) - \theta_{t-1}) \cdot \omega_i(t)$$

where θ is a channel that represents a joint angle and θ_i and ω_i are the desired angle for the joint and the weight for source i , respectively. A partial example of a ViB model using this method to determine the flexion angle of Bob's right elbow is

$$\begin{aligned} & \text{Bob.Right.Elbow.flexion}_t \triangleq \\ & (\theta_{\text{Walk}}(t) - \text{flexion}_{t-1}) \cdot \text{Walk.activation}_t + \\ & (\theta_{\text{Wave}}(t) - \text{flexion}_{t-1}) \cdot \text{Wave.activation}_t + \\ & \dots \end{aligned}$$

where $\theta_{\text{Walk}}(t)$ and $\theta_{\text{Wave}}(t)$ are the desired angle for the elbow joint for the walk and wave behaviors respectively, flexion_{t-1} is a reference to the output of a channel for the previous time-step and Walk.activation_t and Wave.activation_t are references to channel output for the current time-step.

Channel reference naming in the above example is relative to the affected channel. For example, the flexion_{t-1} reference will reference the previous time-step output of the *flexion* channel in the same subsystem as the affected channel (which happens to be the same channel). In cases where there is no ambiguity, naming can be done

relative to any of the parent systems of the affected channel. The *Wave.activation* channel will resolve to the *Bob.Wave.activation* channel since the *Right* and *Elbow* systems do not have a *Wave* subsystem.

Because of the channel accessibility flexibility, it is not required that a referenced channel exist in the model. The output of non-existent channels is assumed to be $\emptyset$. When a model is built, the designer is warned about any referenced channels that are not part of the ViB model.

System Referencing

It is also possible to reference a system within a ViB model which allows the referenced system to act as the output of a function or a channel. Just as it is possible to reference a channel within the same model as mentioned in the previous section, it is also possible to reference a channel within another model using an external channel reference.

Function Composition

A Function represents a mapping from a set of input values to a single output value. In general, the inputs to a function are provided by one or more child functions. For instance, the Binary Multiply Function has two children Functions f_1 and f_2 and defines the mapping $f(t) = f_1(t) \times f_2(t)$.

Some functions require some or all of their children to output certain types of data objects while others do not place restrictions on the type of input. However, in general, a Function does not define a particular function signature (i.e. the domain and range). Instead, the types of the input values that are obtained from the children functions will determine the type of the output of the parent. This is similar to operator overloading in programming languages and reduces the number of different function types that need to be defined.

The architecture provides functions for

- standard arithmetic and trigonometric operators
- comparison and boolean combination operators

- accessing the output history and the delta of a channel or function
- accessing the channel output of another ViB model
- random number generation
- applying orientation changes to vectors
- improved efficiency during evaluation.

Some functions are used to improve execution efficiency by only evaluating certain child functions when necessary. For example, the *weigh* function is a special function that has two children functions, one that provides the weight and another that provides the value to be weighed. For efficiency reasons, the function providing the value will only be evaluated if the weight is not 0. Similarly, the *if* function will choose between one of two children functions to evaluate based on the output of a third condition function.

3.4 ViB Engine

The purpose of the ViB Engine is to evaluate ViB models and allow the output to be connected to other architectures. The architecture of the ViB engine is shown in Figure 3.2. The ViB engine performs two main tasks. It controls model evaluation in coordination with other architectures and supports connections between a ViB model and the entities of other architectures. Also, because of the nature of the ViB model, several optimizations have been made in the ViB engine to improve the efficiency of model evaluation.

3.4.1 Interfacing to Other Architectures

The ViB engine provides facilities to control model evaluation and perform information exchange between a ViB model and other architectures.

Figure 3.2: ViB Engine Architecture

Evaluation Control

The ViB engine coordinates the evaluation of a ViB model. There are three functions that the ViB engine provides to other architectures in order to control model evaluation. The functions are

- trigger the initialization of a newly created ViB model in order to perform model optimizations
- trigger the reset of a ViB model before model evaluation begin
- trigger the evaluation of the model for the next time interval.

Information Communication

The main means of communication between a ViB model and other architectures is through a channel level interface which allows a ViB model to accept input from external sources such as sensors and send output to external targets such as the effectors of the geometry. This interface also allows a ViB model to be used as one or more layers of a layered animation model.

Input to a ViB model is provided through what is referred to as an *external channel*. The ViB engine provides facilities for connecting an external source to a channel such that the channel output is not determined by a channel signal but is instead provided by an external signal that outputs the most recent value provided by the external source. The external source is responsible for sending the appropriate

value to the ViB engine. For instance, sensed information would be forwarded from a sensor to the ViB model through an external channel.

The sending of information from the ViB model to an external target is done by associating a *channel target* with a channel. The output of a channel will be forwarded to the target whenever the channel output changes. Again, the ViB engine provide facilities for registering an external target with a channel of the model. Registering an effector of a geometric model as a channel target will allow the ViB model to change the value of that effector and hence animate the geometry.

3.4.2 Model Evaluation

Because the entities that are used to build a ViB model are first class objects, the behavior of the evaluation process of a ViB model is distributed among these entities. However there are a few aspects of the evaluation process which are presented below.

Channel Evaluation Order

An evaluation order is required between channels and systems in situations where the signal associated with a channel relies on the current output of another channel. The ViB engine will ensure that the channels of a ViB model are evaluated in the proper order so that the referenced channel output is available. The engine will also determine if there are any reference loops that prevent a suitable evaluation ordering of channels. References to a channel's output for any previous time-step do not affect the evaluation ordering.

Within a particular system, the order of evaluation of its children, which includes both channels and subsystems, is determined by ensuring that any child that contains a reference to the current output of a channel of another child is evaluated after that child. For subsystems, all current channel references that occur within any channel signals or subsystems it contains are used to determine this ordering.

This approach to channel evaluation ordering is easily implemented because the evaluation ordering for each system is determined independently of the ordering for parent or child systems. This simplifies the evaluation process greatly. However, it

does place additional restrictions on when current channel output referencing is valid. In most cases, an appropriate system organization will compensate for this.

Channel Output History

Since the output of a channel may be used as input to one or more contribution functions, the output of each channel is recorded so it can be referenced and used to determine the output of any dependant contribution functions. Also, it is common for contribution functions to be based on previous channel output or the change in a channel's output. Therefore, as mentioned in Section 3.3.4, the output of each channel for several previous time-intervals is also recorded. Currently, the ViB engine keeps track of the previous 5 outputs. This is more than we have required for model evaluation but helps in model debugging.

3.4.3 Runtime Issues

There are two main issues we encountered when running the ViB engine: efficiency of model evaluation and debugging of ViB models when undesired behavior arose.

Efficiency

A ViB model contains a set of channels each with its own signal containing one or more contribution functions. The evaluation of every signal within a ViB model on each tick can result in slow execution especially for large models. As a result, we have taken several steps to reduce this inefficiency.

- The ViB engine makes contribution functions and signals as compact as possible by removing redundant contributions and simplifying functions. This is especially important since the generation of ViB models using the techniques described in the chapters that follow can result in functions that can be easily simplified.
- In animation models, it is often the case that several subsystems in the model are dormant for a particular period of time. Therefore, the ViB engine has been

designed in such a way that any system that contains an *enabled* channel will use this channel to determine if the evaluation of its children should take place. The children, including both channels and subsystems, of a system containing an *enabled* channel are evaluated while the output of the *enabled* channel is *true*. If the *enabled* channel of a system has output *false* or 0 for more than 2 ticks, the children of that system are no longer evaluated (except the *enabled* channel itself, which is always evaluated to allow re-enabling). Two ticks are required to allow any channels or subsystems within the system to respond to disabling. The abstractions implemented in Chapter 6 make use of the *enabled* channel to improve evaluation efficiency.

These optimizations increase the scalability of properly designed ViB models. How ViB models are built is described in Chapters 5 and 6.

Debugging

When using the ViB engine to animate a virtual character, it is often necessary to deduce why the character is behaving in a certain way. Indispensable to this process is the ViB model viewer shown in Figure 3.3 which allows the inspection of the systems, channels and signals that make up a ViB model along with the display of the current and previous outputs of channels.

3.5 Summary

In this chapter, we presented the ViB model representation and the tools we implemented to support it. The ViB model representation applies some of the concepts of Synergetics and Dynamic Pattern Theory to a model which supports discrete time-interval evaluation. The following points summarize the ViB model representation.

- The state of a model is defined by a set of variables, referred to as channels.
- Channels are organized into systems which are arranged hierarchically.

Figure 3.3: ViB Model Viewer

- The current value of a channel is determined by applying a delta to the output of the channel on the previous time-step.
- The delta of a channel is determined by combining, or blending, the output of the one or more contribution functions associated with the channel.
- A contribution function is typically a function of the output of one or more channels.
- Channels can output simple data such as scalar values and vectors, and more complex data such as orientations and other ViB models.
- The delta and blend operators are configurable on a channel by channel basis.

The ViB Engine provides the ability to build and evaluate ViB models. It provides an interface to other architectures in order to support input and output of information and evaluation control. It also determines the required channel evaluation ordering to ensure that any channel dependencies are met. Finally, it includes optimizations to improve efficiency, such as sub-system disabling and a model viewer to aid in debugging.

Because the ViB model representation is derived from fields of research that observe naturally occurring behavior, it follows that the model should be capable of representing interesting, natural and realistic behaviors. However, in order to illustrate this, we must build ViB models for particular characters. To do this at the level of abstraction provided by the ViB model would be extremely tedious. Therefore, in the next chapter, we will present the specification technique we will use to build both the higher-level abstractions typically associated with animation models and animation models themselves.

Chapter 4

Using Patterns for Specification

In the previous chapter, we presented the ViB model and supporting architecture. Because the ViB model represents animation models at a low-level, a mechanism for supporting high-level model specification is required along with the supporting architecture for translating these specifications to their corresponding ViB model representations. In this chapter, we first present the requirements of the specification mechanism. Following this, we present the syntax and semantics of our Pattern Oriented Composition Language (POCL) which was inspired by the work of Alexander [2]. In the subsequent chapters, we present how we use POCL to specify ViB models.

4.1 Requirements

In order to promote the reuse of animation model specifications, our specification mechanism must meet the following requirements.

4.1.1 Composition

At the lowest-level, ViB models are constructed by combining a set of channel contributions. A channel contribution associates a contribution function with a channel of a ViB model. Channels in a channel contribution are identified using a system path name from the root of the system hierarchy to the location of the channel. Any

contributions made to the same channel in the set are combined to form a blended channel signal for that channel. The functions themselves are a hierarchical structure that mirrors the function composition hierarchy described in Section 3.3.4. Therefore, we require a specification mechanism that allows the composition of channel contributions for the purpose of creating ViB models. The specification mechanism must support both the composition of simple lists or arrays of entities and as well as entity hierarchies.

4.1.2 Levels of Abstraction

Animation models for virtual characters are typically specified at a higher-level of abstraction than the level presented in the previous section. Therefore, we require a specification mechanism that allows us to specify animation models in terms of the abstractions presented in Section 2.3.1. These include action controllers and other action-selection mechanisms.

At the same time, we do not wish to just implement a specification mechanism that supports the specification and direct translation of higher-level abstractions to their ViB model representations. This is the approach taken by VRML [3] for constructing 3D scene graphs. VRML supports the specification of nodes representing geometric shapes and textures, lighting and transforms for positioning. However, because VRML only offers specification at a single level of abstraction, users of VRML are restricted to using the components that are provided and can only tailor these components in ways anticipated by the creators of VRML.

Therefore, we require the ability to specify different layers or level of abstractions within a single animation model. This ability will allow a designer to use any higher-level abstractions that have been provided or to modify them if required. Modification can be done directly to the abstraction or indirectly through modifications to the lower-level layers on which it is built. The designer can also create new low-level or high-level abstractions as required. The designer's task is further simplified by supporting the specification of these layers within a single specification environment.

4.1.3 Simple

As with the ViB engine, we require that the specification mechanism be simple so that the implementation of the supporting architecture is as well. Therefore, the specification mechanism should only provide facilities that are related to the general task of hierarchical composition. We do require the ability to create low-level ViB model entities within the specification environment. However, this should be done at as low a level as possible in order for the designer to have access to as many levels of abstraction as possible within the same specification environment. In essence, we require a simple specification mechanism that can support a complex set of specifications.

4.2 Pattern Based Specification

In order to meet the requirements presented in the previous section, we view hierarchical composition not as the composition of nodes that represent model entities but instead as the configuration of *patterns* that define relationships between sets of entities. We refer to this process as *Pattern Oriented Composition*. The inspiration for this approach comes from the work done by Alexander on Pattern Languages of Architecture [2, 1]. Just as Alexander's patterns capture general relationships between architectural entities, we use patterns to provide general relationships between the entities that are used to construct animation models. Furthermore, in order to produce models, patterns are arranged in a hierarchical fashion where entities created as a result of smaller-scale patterns act as the entities configured by larger-scale patterns.

Alexander defined a *pattern language* that provides a complete set of instructions for architectural design. In a similar way, we use *pattern libraries* as a means of grouping related patterns together. Static *containment* and *extension* relationships between these libraries help to aid the model creation process by supporting abstraction layering and reuse.

4.2.1 Pattern

In Alexander's work, a pattern is used to describe a configuration or arrangement of entities that is beneficial or desirable when creating architectural designs. In a similar way, we wish to use patterns to describe the relationships between the systems, channels and contribution functions of a ViB model that correspond to or represent higher level behaviors. However, our aim is not just to capture these relationships in order to illustrate how higher-level behaviors can be represented in a ViB model but also to specify these patterns in a format that can be used to automatically generate ViB models representing higher-level behaviors.

Therefore, we view a pattern as a mapping from a set of input entities¹ to a resulting entity that represents the arrangement prescribed by the pattern. Thus, a pattern can be thought of as a function. However, a pattern should not be thought of as an algorithm or computation for determining its result from the set of inputs. Instead, it defines a compositional relationship that constructs the result using the provided inputs as the building blocks.

The nodes in a VRML specification match this interpretation of a pattern. Each node has a set of fields that are used to provide the node with the sub-nodes and data values used to create the desired 3D scene component. Figure 4.1 shows the contents of a VRML file representing a red sphere whose local coordinate system is translated so its center is 3 units above the origin. This example shows example nodes of several node types: Transform, Shape, Appearance, Material and Sphere. Each uses the attributes provided through the named fields to create the appropriate entity.

Just as VRML nodes are used to construct 3D scene graphs, we use patterns to construct ViB models. A VRML specification defines a static arrangement of scene graph components. Some of these components, such as interpolators, may have dynamic characteristics that allow scenes to be animated but the dynamic evaluation of these components is provided by a separate facility, namely a VRML browser. In a similar way, we use patterns to construct ViB models that can then be imported and evaluated by the ViB engine.

¹We use the term entity to refer to the material, components or objects used to construct a model


```
#VRML V2.0 utf8

Transform {
  translation 0 3 0
  children [
    Shape {
      appearance Appearance {
        material Material {
          diffuseColor 1.0 0.0 0.0}}
      geometry Sphere {radius 0.5}}}]}
```

Figure 4.1: Example VRML Node Hierarchy

4.2.2 Entities

How we use patterns to build ViB models is dependant on the *entities* that constitute these models. The structure of a ViB model is derived from the channels and systems that are contained in the model. However, we do not use these entities to construct ViB models. Instead we build models from the lowest level entities in the model, namely contribution functions and channel identifiers. We take this approach because one of the features of a channel is that its output is determined by blending contributions from multiple potentially independent sources and composing ViB models from channel contributions supports this.

The composition of entities using patterns need not be specific to the construction of ViB models. Therefore, except for entities related to pattern oriented composition in general, the creation of entities is not supported directly by our specification language. Instead, the entities and relationships that are specific to a particular domain can be incorporated into our specification language as pattern primitives, which are described in more detail in Sections 4.3.5 and 4.4.6.

4.2.3 The Pattern Oriented Composition Language

The goal of our specification language is to provide a syntax and semantics that allows pattern oriented composition. A further goal is to simplify the syntax and semantics of the language by providing facilities at as low a level as possible. In the following section we present the syntax of POCL. Then, in Section 4.4, we present the semantics of how patterns are used to create and compose entities.

4.3 Syntax

The syntax we use to specify patterns is derived from VRML. However, it has been tailored using concepts borrowed from Alexander's patterns [2]. Because the language deals with pattern oriented composition in general, POCL has a fairly small number of syntactic constructs, which we present below.

4.3.1 Entities

Patterns are concerned with the creation and composition of entities. In POCL, an entity can be represented as

- a literal representing a number or string

```
1
5.5
"Walk"
```

- a combination of entities using common binary operators

```
entity1 * entity2 + entity3
```

- a list of entities denoted using square brackets

```
[entity1 entity2 ... entityN]
```

- a pattern configuration as described in the next section
- a reference to a variable.

Pattern definitions, libraries and contexts, which are presented below, are also special types of entities.

4.3.2 Pattern Configuration

The purpose of a pattern configuration is to provide input variables to a pattern. Therefore, a pattern configuration consists of a set of variable bindings and identifies the pattern to which the bindings are to be made available. The general syntax of a pattern configuration is

```

PatternName{
  variable1 = entity1
  variable2 = entity2
  ...}

```

where each entity could be any of the entity representations presented in the previous section including other pattern configurations. The following example is a configuration of the pattern named P1.

```

P1{
  x = 10
  y = P2{w = 3.5}
  z = [P3{text = "syntax"} P3{text = "example"}]}

```

In the above configuration, *x* is bound to a literal, *y* is bound to a smaller-scale pattern configuration and *z* is bound to a list of two configurations of the P3 pattern.

There are two supported formats for pattern configurations. The above format is referred to as the *node* style since its format is similar to that of VRML. The following format is referred to as the *function* style.

```

PatternName (entity1, entity2, ...)

```

The function style configuration format is provided to allow for a more concise specification. As we will see in Section 4.4.4, the function style is an extension of the node style.

```

P4(10, P3{text = "function"})

```

The above configuration of P4, in the function style, has two input parameters. The first is a literal while the second is a configuration of P3, in the node style.

4.3.3 Pattern Definition

Pattern definitions allow for the creation of patterns that can be configured. The syntax of a pattern definition is

PatternName = **Pattern** *entity*

The body of a pattern definition is simply an entity that is typically a composition of smaller-scale entities, some of which may be variable references. In the following example, the definition of pattern P1 is a configuration of P4.

```
P1 = Pattern P4{
  u = x * y
  v = P5(x, z)}
```

The input variables of a pattern are implicit in the body of the definition. The above definition contains references to the variables x, y and z.

Alternately, a pattern definition can have a pattern signature which allows it to be configured using the function style.

PatternName = **Pattern signature**(variable1, variable2, ...) *entity*

The header of the pattern definition for pattern P4 could be

```
P4 = Pattern signature(u, v) ...
```

which allows for the following alternative definition of P1.

```
P1 = Pattern P4(x * y, P5(x, z))
```

As we will see in Section 4.4, the pattern signature's sole purpose is to allow function-style configurations and, therefore, the signature does not necessarily indicate all input variables.

4.3.4 Variable Reference

The syntax of variable identifiers is similar to that of many programming languages. A variable identifier starts with a letter and contains any number of letters and digits. As was stated above, a variable can be used to represent an entity within the body of a pattern definition. A pattern can be configured without providing all input variables.

Therefore, the syntax of a variable reference includes the optional specification of a default value via the special operator `:=` should one not be provided in a configuration.

```
variable:=(entity)
```

The following definition of P1 provides a default value of 1 for y and a configuration of P3 as the default value for z.

```
P1 = Pattern P4{
  u = x * y:=(1)
  v = P5(x, z:=(P3{text = "default"}))}
```

There are also special variable qualifier keywords, *super* and *outer*, whose use are related to the semantics of variable value determination and are therefore explained in Section 4.4.5.

4.3.5 Pattern Primitive

The syntax of POCL only supports the direct specification of low-level entities such as numbers, strings and lists, and entities associated with patterns, such as pattern definitions, contexts and libraries. Support for other entities is provided using pattern primitives.

```
PatternName = Primitive signature(variable1, variable2, ...) primitivIdentifier
```

This syntax is similar to that of a pattern definition except that the body is an identifier which indicates to POCL a function or procedure that has been externally provided. This is similar to the approach taken by object-oriented languages such as Smalltalk and Java that use primitives or dynamic link libraries (DLL) to provide low-level facilities that are more efficiently implemented in a lower-level language.

4.3.6 Context

Some of the pattern primitives provided for POCL make use of a context entity. A context is a means of providing a set of variable bindings to a pattern primitive. The syntax for specifying a context is similar to a pattern configuration with the pattern name replaced by the Context keyword.

```

Context {
  variable1 = entity1
  variable2 = entity2
  ...}

```

4.3.7 Pattern Library

A pattern library contains a set of pattern definitions and can also contain other libraries. Library management is provided through the use of a POCL browser application. However, we use the syntax presented below to describe a library organization.

```

Library A {
  Library B {
    P1 = Pattern ...
    ...}
  Library C extends B {
    P1 = Pattern super.P1 {...} }}

```

We will present the relevance of library structuring in Section 4.4.5. For now, we will summarize the above organization by saying that library A contains B and C, library C is an extension of library B, and library B and C both contain a pattern definition for pattern P1. Furthermore, the definition of pattern P1 in library C extends the definition of P1 provided by library B by using the *super* reference qualifier. Another reference qualifier, *outer*, is also available and will be described in Section 4.4.5.

4.4 Semantics

The purpose of the pattern syntax presented in the previous section is to allow the specification of models for virtual characters. Models specified using patterns must be translated to the model representation or exchange format of the target domain. We refer to this translation process as pattern resolution. In this section, we introduce the general semantics of POCL for pattern resolution. More specific examples will be given in later chapters.

4.4.1 Pattern Resolution

Although patterns map a set of inputs to an output, pattern resolution is not like a function call but is instead a form of macro expansion that translates a pattern definition into an entity. Before we describe the exact nature of pattern resolution, we will give an overview of the pattern resolution process.

Pattern Configuration

A pattern configuration is used to trigger the resolution, or expansion, of a pattern definition. When a pattern is configured, any variable references that appear within the corresponding definition are replaced by the entities bound to those variables in the configuration. For example, consider the following definition of the P1 pattern.

```
P1 = Pattern P2{
  v = x
  w = P3{x = P4{z = y}}}
```

As stated previously, the inputs of a definition are implicit in the body of the definition. In the above definition of P1, the variable x represents the entity to be assigned to v in the configuration of P2 while y represents the entity to be assigned to z in the configuration of P4. Therefore, the variables x and y are input variables of the pattern P1. A pattern configuration's purpose is to provide values for these variables.

```
P1{
  x = entity1
  y = entity2}
```

In the above configuration of P1, let *entity1* and *entity2* be place holders that represent the input entities of P1. We will use these place holders to illustrate how substitution takes place. When P1 is configured, and hence expanded, whatever is bound to x and y replaces the references to these variables in the body of the definition of P1 and then the body of the definition replaces its configuration. Therefore, the above configuration of P1 would expand to the following configuration of P2.

```

P2{
  v = entity1
  w = P3{x = P4{z = entity2}}

```

This process is repeated for P2 and for any configurations that appear in the definition of P2. This recursive expansion ends when a pattern definition is encountered that does not contain any further pattern configurations. For instance, if the definition of P2 was

```
P2 = Pattern [v w]
```

then the configuration of P1 would eventually expand to a list containing two elements, the first of which would be *entity1* while the second would be the expansion of the nested configuration $P3\{x = P4\{z = \textit{entity2}\}\}$ as described in the next section.

Nested Configurations

The definition of P1 in the previous section and its expansion illustrates how pattern configurations can be nested. The following sub-pattern that results from the expansion of P1 is an example of a nested pattern configuration.

```
P3{x = P4{z = entity2}}
```

To simplify the explanation of the semantics of pattern resolution in the previous section, we described the process from the standpoint of the outermost configuration. However, the expansion process actually starts at the innermost configuration and proceeds to the outermost one. Therefore, the configuration of P4 would be completely expanded and the result assigned to the variable x before the expansion of P3 would begin. The reasons for this ordering is due to the variable scoping mechanism which is described below in Section 4.4.2.

Binary Operators and Lists

The use of standard binary operators for building and combining entities is supported by POCL. These include the common arithmetic operators (+, -, *, /, &, |) along with the comparison operators (==, ~=, <, <=, >, >=) and a special operator (@) for creating 2D and 3D vectors. Binary operators are viewed as pattern configurations

with two variables. Therefore, the arguments of a binary operator are expanded before the operator is applied. Similarly, the elements in a list are expanded and the results become the elements of the list.

4.4.2 Variable Scoping

In this section, we describe more precisely how we determine the entities that are substituted for variable references in a pattern definition.

Scoping within Nested Configurations

The variable bindings that appear within a single configuration are order independent. In other words, a configurations does not contain a sequence of variable bindings but instead contains a set of bindings. Consider the following definition of P1.

```
P1 = Pattern P2{
  x = y
  y = x}
```

The second binding in the configuration of P2 binds the variable y to the entity provided by x. However, this x is the variable bound in the configuration of P1, not in the first binding in the above definition. Therefore, the following configuration of P1

```
P1{
  x = entity1
  y = entity2}
```

would expand to the following configuration of P2.

```
P2{
  x = entity2
  y = entity1}
```

where x and y are replaced by the entities bound in the configuration of P1 without regard for other bindings within the same configuration.

This scoping holds true for nested configurations as well. Therefore, if the definition of P1 were changed to

```

P1 = Pattern P2{
  x = y
  y = P3{x = P4{w = x}}
}

```

Then the reference to *x* in the configuration of *P4* is still a reference to the entity bound in the configuration of *P1* shown above, not the entity bound to *x* in the configuration of *P2* within the definition of *P1*.

Scoping During Configuration Expansion

During pattern resolution, the expansion of a pattern configuration consists of two phases. The first phase is the resolution of the variable bindings of the configuration and the second is the expansion of the referenced pattern to its definition.

Consider the following configuration of *P5*.

```

P5{
  x = 10
  y = P4{w = 3.5}}

```

The resolution of the variable bindings of this configuration results in the creation of a *resolution frame* containing two bindings.

```

x = 10
y = entity1

```

The first frame entry simply binds the value 10 to the variable *x*. The entity bound to *y* is the result of recursively resolving a smaller-scale configuration. For now we will assume that the result of the resolution of the *P4* configuration is an entity that we represent as *entity1* (Examples that involve actual entities will be presented in the next chapter).

The second phase involves the substitution of the corresponding pattern definition for the pattern being configured and the substitution of corresponding entities for any variables references that appear in the body of the definition.

```

P5 = Pattern P2{
  y = x + y
  z = [P6{text = "syntax"} P6{text = "example"}]}

```

In this case, *P5* expands to the following configuration of *P2*.

```

P2{
  y = 10 + entity1
  z = [P6{text = "syntax"} P6{text = "example"}]}

```

The above described configuration resolution process is then reapplied to the resulting configuration. The smallest scale pattern definitions either make use of literals or primitive patterns to create entities. Therefore, the expansion continues until such a definition is encountered.

The scope of a variable during resolution is not restricted to a single level in the expansion. To illustrate this, consider the following resolution frame that results from the resolution of the above configuration of P2.

```

y = 10 + entity1
z = entity2

```

The entity bound to *z*, *entity2*, would be a list containing the two entities that resulted from the two configurations of P6.

The following definition of P2 references the variables *x*, *y* and *z* (P2 was previously referenced but not defined).

```

P2 = Pattern P3(x, y, z)

```

The values for *y* and *z* are obtained from the frame associated with the configuration of P2. The value for *x* is not contained in this frame so is obtained from the frame associated with the configuration of P5 that led to the configuration of P2. Therefore, the above configuration of this definition expands to the following.

```

P3(10, 10 + entity1, entity2)

```

The above example illustrates the scoping mechanism used during pattern resolution. We refer to the environment which provides the values for variables as the *resolution context*. The resolution context consists of a hierarchy of frames corresponding to the chain of configurations in an expansion. Figure 4.2 shows the resolution context for our example. Variable lookup begins in the frame corresponding to the configuration being expanded. If a variable reference that appears within the definition is not bound in this frame, then lookup proceeds up the hierarchy until either a binding is found or the root level of the expansion is reached.

Figure 4.2: Resolution Context

This scoping mechanism amounts to dynamic scoping. Programming languages that use dynamic scoping are typically difficult to understand. However, the pattern resolution process is not concerned with creating and executing a program. It is concerned with the composition of entities. There are still issues with dynamic scoping that must be overcome. One problem is that of naming collisions between larger-scale patterns and smaller-scale patterns that use variables of the same name for different purposes. Care must be taken to pick variable names that reflect the full nature of the variable.

It should be noted that a variable binding in a configuration has no effect on bindings made to a variable of the same name by higher-level configurations but instead obscure this higher-level binding as shown in Figure 4.2 for the y variable.

4.4.3 Pattern Definitions as Entities

The determination of the definition for a pattern being configured uses the same scoping mechanism as described in the previous section. In other words, a pattern is simply a special type of entity that can be configured using the configuration syntax. For example, consider the following definition of $P1$.

```

P1 = Pattern P2{
  u = p2{
    x = 1
    y = P3{}}
}

```

Notice that the above definition makes use of the variable `p2` as if it were a pattern. Now consider the following configuration of `P1`.

```

P1{
  p2 = Pattern x + y
}

```

Any configurations of `p2` that appear in the definition of `P1` expand to the definition bound to `p2` in the above configuration. It should be noted that the variable references `x` and `y` that appear in the definition of `p2` are substituted when the `p2` pattern is configured and not when it is defined. Since the name of a pattern being configured is treated as a variable reference in the resolution context, a means to manage the process of providing proper pattern definitions is required. This ability is provided by using pattern libraries, which are described in Section 4.4.5.

4.4.4 Pattern Signatures

A pattern definition may have a signature that allows the pattern definition to be configured using the function-style configuration.

```

p1 = Pattern signature(x, y) x + y

```

The existence of a pattern signature and the use of a function style configuration does not change the variable scoping rules. Instead, the ordered arguments in a function-style configuration are bound to the corresponding variables of the signature definition in a resolution frame. Therefore, the pattern configuration

```

p1(10, P4{w = 3.5})

```

is equivalent to

```

p1{
  x = 10
  y = P4{w = 3.5}}

```

A pattern signature does not restrict what variable references appear in a pattern

definition and it does not prevent the pattern from being configured using the node style. The signature's only purpose is to allow the specification of function-style configurations when desired. Also, not all variables that appear in a signature need to be provided in a function style configuration. The values provided in the configuration will be mapped to the corresponding variable name and any additional variables in the signature will be ignored. For instance, the configuration

```
p1(10)
```

does not provide a value for *y*. Therefore, the variable *y* will not be bound in the resulting frame. In such a case, either the definition of *p1* would provide a default value for *y* or a configuration higher in the resolution hierarchy could provide a binding for the variable.

4.4.5 Pattern Libraries

Pattern libraries are used to group a set of related patterns together. In addition, they act to determine how and when patterns are loaded into the resolution context. The following sections describe how libraries are used to organize patterns.

Configuring Libraries

A library contains a set of patterns. For example, the following library *A* contains definitions for patterns *A*, *P1* and *P2*.

```
Library A {
  A = Pattern P1 {
    u = P2 (x)}
  P1 = Pattern ...
  P2 = Pattern signature(z) ...
}
```

A library can be configured in the same way as a pattern definition as long as it contains a pattern definition of the same name as the library, which we refer to as the library's *entry pattern*. In the following configuration of *A*, let *entity1* be a placeholder representing an arbitrary entity.

$A\{x = \text{entity1}\}$

Given the definition of A contained in library A, this configuration expands to the following.

Pattern P1 {
 $u = P2(\text{entity1})\}$

The references to patterns P1 and P2 in the above configuration expand to the corresponding definitions found in library A. This is due to the fact that, when a library is configured, the definitions contained in the library are added to the resolution context before the library's entry pattern is resolved and expanded. This is shown in Figure 4.3.

Figure 4.3: Configuring a Library

Patterns as Input

A pattern library is often used to provide a set of patterns corresponding to a particular layer of abstraction. In such a case, a pattern library acts as an organizational mechanism to both group the related patterns together and to allow separate specifications using these patterns to generate the corresponding entities. To illustrate this, we will consider the following *AbstractionLayer* library which contains the *AbstractionLayer* pattern definition along with several other pattern definitions that

represent the abstractions provided by the library.

```
Library AbstractionLayer {
  AbstractionLayer = Pattern specification {}
  Abstraction1 = Pattern ...
  Abstraction2 = Pattern ...
}
```

The input to the AbstractionLayer pattern is a variable called “specification” which is intended to be a pattern containing configurations of one or more of the other patterns, namely “Abstraction1” and “Abstraction2”, provided by the library.

```
AbstractionLayer{
  specification = Pattern Abstraction1 {
    x = Abstraction2{}}}
```

Figure 4.4: Using a Pattern as Input

The above configuration simply expands to the configuration of the provided specification pattern which in turn expands to the configuration of Abstraction1. The expansion of the above configuration along with the resulting resolution frames is shown in Figure 4.4. As previously stated, when a library is configured, the pattern definitions contained in the library are added to the resolution context before the library’s entry pattern is expanded. Consequently, the configuration of Abstraction1 at the bottom of Figure 4.4 will be further expanded using the Abstraction1 and

Abstraction2 pattern definitions provided by the AbstractionLayer library. This illustrates how libraries can be used to support the separation of abstractions from the specifications that use them.

In addition to providing an input specification pattern, users of a library can also provide, as input, the lower level abstractions that are used by a library. Consider the following partial definition of Abstraction1.

```
Abstraction1 = Pattern LowerLevelAbstraction{...}
```

This definition makes use of a pattern, LowerLevelAbstraction, that is not defined in the AbstractionLayer library.

```
AbstractionLayer{
  LowerLevelAbstraction = Pattern ...
  specification = Pattern Abstraction1 {
    x = Abstraction2{}}}
```

Figure 4.5: Lower Level Abstractions as Input

The expansion of the above variation of our original configuration of the AbstractionLayer library is shown in Figure 4.5 and illustrates how a higher-level configuration can provide the required pattern representing a lower-level abstraction. As we will see in the chapters that follow, the lower level abstractions used by a library

are not usually specified in the manner shown above but are instead implemented as other libraries. In this way, libraries can be used to layer abstractions.

Library Containment

As can be seen in the previous two library examples, a pattern library can also act as a pattern definition. As such, it can be contained in a library just as a pattern definition can, thus supporting library nesting. This gives rise to a pattern containment hierarchy.

```

Library A {
  A = Pattern B {
    specification = Pattern P1 {
      u = P2 (x)}
  P1 = Pattern ...
  Library B {
    B = Pattern specification {}
    P2 = Pattern ...}
}

```

Consider the following configuration of the above library.

```
A{x = entity1}
```

Before we consider the expansion in detail, note that P1 is defined in library A whereas P2 is defined in library B. These respective definitions are found specifically because of the dynamic scoping mechanism provided. Consequently, the library containment hierarchy has a large influence on the resolution hierarchy.

Now consider the detailed expansion of the above configurations in Figure 4.6. Intuitively, A expands to a configuration of B which set up the input “specification” variable. B then expands “specification” which, in turn, expands P1 as defined in library A. When P2 is subsequently expanded (not shown), it will be the definition of P2 from library B.

Library Extension

A library can be extended by another library. This is similar to inheritance in object-oriented programming languages. Consider the following example in which library B

Figure 4.6: Library Containment

extends library A.

```
Library A {
  A = Pattern specification {}
  P1 = Pattern P2 {...}
  P2 = Pattern ...}
```

```
Library B extends A {
  B = Pattern A {}
  P1 = Pattern super.P1 {
    x = P3 (x)}
  P2 = Pattern ...
  P3 = Pattern ... }
```

As in object-oriented inheritance, library B is said to inherit the patterns contained in A. Library B inherits the patterns A, P1 and P2 from library A. Furthermore, a pattern defined in library B with the same name as a pattern in library A overrides the inherited pattern. Library B overrides both the P1 and P2 patterns. The *super* keyword can be used to reference an inherited pattern. The definition of P1 in library B references the definition of P1 in library A using the super qualifier.

To illustrate how inheritance affects expansion, consider the following configuration of library B.

```
B {
  x = entity1
  specification = Pattern P1 (x)}
```

The expansion of the above configuration is illustrated in Figure 4.7. Intuitively, the configuration of B sets up “x” and “specification” and expands to A which then expands “specification”. This in turn expands P1 (defined in library B) which expands to P1 (defined in library A) which expands P2 in library B (whose details are not shown).

Figure 4.7: Library Extension

To repeat the explanation in more detail, after the creation of the resolution frame for the bindings made in the configuration of B, the resolution frames for library A and B are created. Notice that the frame for library B is a child of the frame for A. Once these frames are created, the configuration of B is expanded to a configuration of A because of the definition of B in the lowest frame. The configuration of A expands

further to a configuration of the supplied specification pattern using the definition of A from the second lowest frame since the lowest frame does not contain a binding for A. Finally, A expands to a configuration of the supplied specification from the third lowest frame since neither of the bottom two contain a binding for it.

The specification configuration expands to $P1(entity1)$. Again, lookup for P1 begins in the lowest frame and, thus, expands to $super.P1(entity1)$. The lookup for “super.P1” begins at the frame corresponding to the library that library B extends, namely library A. Therefore, this reference expands to $P2\{v = entity1\}$. The expansion would continue using the definition of P2 from library B.

References that are qualified using the super keyword do not follow the usual dynamic scoping rules. Instead, lookup begins in the resolution frame corresponding to the extended library and lookup proceeds up the resolution hierarchy in a manner that corresponds to the library extension hierarchy. There are some additional aspects of super that are related to pattern library containment. If super is used in a library that does not extend another library or references a pattern that is not provided by an extended library, then lookup will proceed through the library containment hierarchy. For instance, consider the following pattern organization.

```
Library A {
  Library B {...}
  Library C extends B {
    D = Pattern super.D }}
```

Lookup for the super.D will begin in the resolution frame corresponding to B. If this library does not contain the pattern D, then lookup would proceed to the containing library A.

The super qualifier can also be used to allow a library to extend another library of the same name in the same way as pattern definitions can, as illustrated below.

```
Library A {
  Library C {...}
  ... }

Library B extends A {
  Library C extends super.C {...}
  ... }
```

In the above example, the C library defined in B overrides the C library in A due to the fact that B extends A.

Pattern Overriding

In some cases, it is desirable to override a pattern in the dynamic resolution context hierarchy. This capability is provided using the special keyword *outer* which acts as a reference qualifier. Consider the following library definition.

```
Library A {
  A = Pattern specification {}
  P1 = Pattern P2 ...
  P2 = Pattern ...
}
```

In the following configuration of the above library, the outer qualifier is used to override the P2 pattern only for the given configuration of P1.

```
A {
  y = entity1
  specification = Pattern P1 {
    P2 = Pattern outer.P2 {
      x = y}}}
```

Figure 4.8 shows the expansion that results from the resolution of the above configuration. Intuitively, the configuration of A provides a value for “y” and “specification”. This leads to the expansion of “specification” which expands to P1 and then P2 (due to the definition of P1 in library A). P2 expands to outer.P2 which expands to P2 (whose details are not shown) as provided by library A.

More specifically, the use of the outer qualifier causes lookup for the reference to begin in the parent frame of the one in which the innermost pattern definition containing the outer reference is bound. Therefore, once the above configuration has been expanded to a configuration of P2, this configuration will expand to the version of P2 bound in the lowest frame in Figure 4.8. Furthermore, this version references the P2 in the second lowest frame by using the outer reference qualifier. Note that if outer was not used, this pattern definition would reference itself which is invalid. If no binding of P2 existed in this frame, lookup would proceed up the resolution hierarchy.

Figure 4.8: Dynamic Pattern Overriding

Note that this behavior is different than that of `super` since `super` causes lookup to begin in the parent library of where the pattern containing the qualified reference is defined whereas `outer` causes lookup to begin in the parent resolution frame of where the containing pattern definition is bound. The Closure pattern primitive has similar characteristics to the `outer` keyword.

4.4.6 Entities and Pattern Primitives

As previously stated, the creation of domain specific entities, such as those for the ViB model, is not supported by the syntax of POCL. Instead, pattern primitives are used to allow the creation and low-level combination of these entities. For instance, to draw an analogy with home construction, pattern primitives would be provided to create (or supply) lumber and nails and to drive the nails into the wood. Larger-scale patterns would use these primitives to place the pieces of lumber and position in the nails properly to create a wall or a door.

Like a pattern definition, a pattern primitive configures a set of input variables. However, the configuration of these variables is not specified using POCL but is instead specified using a general purpose programming language. When a pattern primitive is resolved, it has access to the variable bindings made by higher-level configurations in the resolution context. The primitive can then access, from this context, the variables it requires to either create a new entities or combine a set of entities (which would be provided through one or more variables). The entities created by these primitives act as the material that is used to construct models and as an exchange format between the pattern specification environment and target domain.

The syntax of POCL does support the creation of general entities such as numbers, strings and lists. There is also some syntactic support for combination of entities using simple binary operators. The results of using these operators depends on the entities involved. For instance, for contribution function entities, the arithmetic operators are not evaluated during pattern resolution but instead act to create a contribution function that, when evaluated by the ViB engine, performs the corresponding arithmetic combination.

4.5 POCL Tools

The Pattern Oriented Composition Library (POCL) environment provides support for the specification and use of patterns definitions and libraries. The two main tools provided by the environment are the POCL Library Browser for creating and managing pattern libraries and the POCL compiler for generating ViB models from pattern specifications.

4.5.1 POCL Browser

The library structure provided through the extension and containment relationships is important in creating and using abstraction layering. Consequently, we have provided a browser, shown in Figure 4.9, for navigating the library structure, for creating libraries and pattern definitions and for saving and loading libraries from files. This

Figure 4.9: POCL Library Browser

tool was instrumental in the creation and management of the pattern libraries required to generate ViB models.

Parsing

The POCL Browser provides a simple parser for converting pattern definitions into the internal representation that is used by the POCL Compiler. Since pattern oriented composition deals strictly with composition, there are a small number of objects that the browser and parser must deal with. These include pattern libraries, pattern definitions, two styles of pattern configuration, variable binding and referencing, string and number literals, arrays and the supported binary operators. Conversion of pattern libraries and definitions specified using the syntax presented in this chapter into the corresponding internal representation is a straightforward process.

The parser is greatly simplified because it does not have to deal with the specification of the low level relationships such as entity creation and combination. The parser

does provide a mechanism to link a pattern definition to a pattern primitive using the syntax shown in Section 5.1.4. However the mechanism by which the primitives are used to create or combine entities is discussed below in Section 4.5.2.

Cross-Referencing

The POCL browser also makes use of the parsed pattern libraries and definitions in order to provide cross-referencing tools. Designers can query the browser for definitions and references to both pattern libraries and definitions. This tool is helpful during creation because it allows designers to determine and investigate larger-scale and smaller-scale patterns that are of relevance and to look at examples of how a particular pattern is used.

4.5.2 POCL Compiler

The POCL compiler performs pattern resolution using the pattern libraries and definitions loaded into the POCL Browser. The purpose of the compiler is to create models for use with the ViB Engine.

Creating a Model

At the root of the compilation process is a pattern configuration that represents the overall model of a virtual character. How this is accomplished is presented in the next chapter.

Implementation

The compiler is basically a macro expansion facility that supports dynamic scoping in the resolution hierarchy and static scoping through the pattern libraries. Therefore, its implementation is only concerned with context management during resolution. The compiler architecture is provided with object classes for representing pattern definitions and libraries and for constructing the resolution context during pattern resolution. The resolution context classes provide the necessary mechanisms for resolving variable references, including those involving super and outer, and for pattern

signature retrieval. The amount of code required to implement the compiler is small (approximately 25 classes) due to the small syntax and semantics used.

The compiler does provide some support for error handling during resolution. Facilities are provided to halt the resolution process within a particular pattern definition to ensure that the resolution context in which the pattern is being resolved has been properly formed by larger-scale patterns. Trace back facilities also exist to aid in situations when required variables are not present.

The compiler is greatly simplified through the use of pattern primitives which perform the task of creating and combining model entities. It is really these primitives that define what can be done using POCL. Therefore, suitably designed low-level primitives are essential to the success of this approach. The number of primitives for a model depend on the entities in the target model. For the ViB model, the number of primitives is small (approximately 20) due to the simplicity of the model. However, creating the proper relationships between the channels in a ViB model requires additional primitives that aid in the composition process by providing low-level relationships that, although applicable to ViB model creation, are really related to composition in general. These low-level primitives are discussed in the next section.

4.5.3 Primitive Patterns

During pattern resolution, the POCL Compiler deals strictly with resolution context management and depends on pattern primitives to create and arrange entities. When resolved, a pattern primitive accesses variables from the current resolution context in order to create the appropriate entity. In essence, pattern primitives create the material from which models are made. The POCL compiler simply manages the process that allows these entities to be combined with other entities.

From an implementation standpoint, a primitive is an object that is imported into the POCL compiler. Interfacing between a primitive and the compiler is a two step process. First, the compiler retrieves the pattern signature associated with the primitive in order to properly bind the input parameters of any function-style configurations of the primitive to the proper variables. Second, when resolved as part of

a pattern configuration, the primitive is given access to the current resolution context represented as a first class object. The primitive can access any of the variables available in the context and from these variables, create and return the appropriate entity.

There are several pattern primitives that deal with composition in general. Others aid in concise specification. The POCL environment provides these primitives along with others used to create custom contexts, resolve pattern definitions in these contexts and support basic list operations. As we will see in the next chapter, there are several sub-architectures that are combined to create a complete virtual character and each of these has a corresponding set of pattern primitives. The specifics of these primitives will be presented as they are used in the chapters that follow.

4.6 Summary

In this chapter, we presented the syntax and semantics of POCL, the language we use to specify ViB models. Derived from Alexander's work on patterns, POCL is concerned with the composition of entities instead of algorithm specification.

The syntax of POCL is similar to that of VRML. A pattern configuration provides the input to a pattern definition as a set of variable bindings. POCL supports both a node-style configuration, which explicitly names the variables of a configuration, and a function-style configuration, which uses a pattern signature to map an ordered set of parameters to an ordered set of variables, much like a function call in most programming languages.

The semantics of POCL are similar to macro expansion. A pattern configuration expands to the pattern's definition and the variable bindings in the configuration are substituted for any variables that are referenced in the definition. However, dynamic scoping is used during expansion so that any variables that are not provided by the immediate configuration of a pattern will be obtained from the chain of larger-scale expansions that lead to the current expansion.

The lowest-level of any expansion are pattern primitives. A pattern primitive is an

externally implemented procedure for generating the desired entity. The entities created by pattern primitives are the material used to construct models for a particular domain.

Pattern definitions can be grouped in pattern libraries. A pattern library is actually just a special form of pattern definition. A library can contain pattern definitions or smaller-scale libraries, thus forming a library containment hierarchy. Libraries also support a form of extension similar to inheritance in object oriented programming languages. We have provided tools to create and compile POCL libraries.

In the next chapter, we look at the pattern primitives associated with the ViB model and also discuss the high-level library organization we developed in order to build virtual characters.

Chapter 5

Specifying ViB Models as Patterns

The main goal of this chapter is to relate pattern oriented composition to ViB model and virtual character creation. Therefore, in this chapter, we introduce the entities used to construct ViB models and the primitive patterns that allow these entities to be imported into POCL. At the same time, we reiterate the syntax and semantics of our specification language using concrete examples involving these pattern primitives. We also illustrate how pattern libraries provide abstraction layering and support reuse. We do this by first introducing the architecture we use to represent complete virtual characters and the sub-architectures that constitute this architecture, one of which is the ViB Engine. We then present the libraries that contain the pattern primitives of these sub-architectures and describe how these libraries are arranged and used to generate the desired models.

5.1 Creating ViB Models

We are going to illustrate how to build ViB models using patterns with some simple examples. First, we will describe the entities and primitive patterns used to construct ViB models. Before we do this, it is important to reiterate that the specifications we use to build ViB models are not executed themselves. Instead, they are used to build models that can be executed by a separate architecture. Therefore, the entities used to construct ViB models are model components that act as an exchange format between

the POCL environment and the ViB engine. It is the ViB engine that evaluates the resulting model. Patterns are strictly used for composition.

5.1.1 Entities

As was stated in Section 4.2.2, ViB models are constructed from the lowest level entities that constitute the model. These entities include

- the *channel identifier* entity which is used to identify a channel in a ViB model. A channel is identified using a list of names representing a channel name that is qualified using a system path name from the root of the system hierarchy to the system containing the channel.
- the *contribution function* entity which represents a function that can be contributed to a channel or combined with other contribution functions using function composition. The most important contribution function is a *channel output* function which allows the output of a channel to be used as a contribution function.
- the *channel property* entity which represents one of the properties of a channel such as the delta and blend operators or initial output value.
- the *channel contribution* entity which associates a contribution function with a channel identifier and represents a contribution made to that channel. The channel contribution entity is also used to associate a channel property with a channel.
- the *ViB model* entity which represents the combination of a set of channel contribution entities. The channel identifiers of the channel contributions are used to create the system hierarchy and channels. A channel's signal is created by blending the contribution functions of all contributions whose channel identifiers match that of the channel.

Higher-level entities are represented as a set of channel contributions. For instance, an action entity is represented by a set of channel contributions containing

the contributions to the channels representing the effects of the action on the DOFs of the character along with some additional contributions that control the activation of the action.

5.1.2 Primitive Patterns

The ability to create the entities of a ViB model is provided by a set of primitive patterns. There are three main patterns that are used to construct ViB models.

- The ChannelContribution pattern converts a channel identifier and a contribution function into an channel contribution entity.

`ChannelContribution = Primitive signature(channel, contribution) ViBChannelContribution`
where the identifier `ViBChannelContribution` represents an external object that provides the functionality for creating channel contributions.

- The channelOutput pattern creates a contribution function entity that allows the output of a channel to be contributed to other channels. The inputs to the channelOutput pattern are a channel identifier and an offset indicating whether the current (0) or previous (-1) output of the channel is to be used as the contribution. If an offset is not provided, the default of -1 is used.

`channelOutput = Primitive signature(channel, offset) ViBChannelOutput`

- The ViBModel pattern combines a set of channel contribution entities in order to create a ViB model entity.

`ViBModel = Primitive signature(name, contributions) ViBModelEntity`

There are also several pattern primitives, such as the delta pattern used below, for creating the different types of contribution functions mentioned in Section 3.3.4. There are also pattern primitives related to pattern composition in general which will be introduced as they are used.

In the next section, we will provide example configurations of each of the three main patterns, ChannelContribution, channelOutput and ViBModel, which expand

to the underlying primitives in order to create the entities presented in the previous section.

5.1.3 Pattern Configuration

In this section, we will demonstrate how patterns are used to create a ViB model entity that can be evaluated by the ViB engine. To begin, consider the following pattern configuration.

```
ChannelContribution{  
  channel = ["location"]  
  contribution = channelOutput(["velocity"], -1) * delta(t)}
```

For now, we will simply view this as a configuration of the ChannelContribution pattern that consists of bindings for the channel and contribution variables. Furthermore, the contribution variable is bound to the result of expanding two smaller scale pattern configurations, channelOutput and delta, and combining the results of these separate expansions using the binary operator *. In the sections that follow, we will describe how the above pattern configuration expands to an entity that can be used as part of a ViB model.

Function Style Configurations

The channelOutput and delta patterns used above are function style configurations. Figure 5.1 shows the expansion of channelOutput configuration into the corresponding contribution function entity. The first stage of the expansion involves the binding of the two provided parameters to the corresponding variables named in the signature of the pattern, which are “channel” and “offset”. The value bound to the variable named channel is a list of strings representing the channel identifier (in this case, the channel is the *velocity* channel).

The next stage is the expansion of channelOutput to its definition followed by the expansion of the ViBChannelOutput primitive. As stated in Section 4.4.6, a primitive can be thought of as an externally implemented function that references certain variables in the resolution context in order to create the appropriate entity.

Figure 5.1: channelOutput Expansion

The ViBChannelOutput primitive uses the values of channel and offset variables in order to create a contribution function entity. In Figure 5.1, the resulting entity, $velocity_{t-1}$, is a contribution function that, when evaluated by the ViB engine at time t , will output the value of the *velocity* channel at the previous time step $t - 1$.

Binary Operators

The use of standard binary operators for building and combining entities is supported. These include the common arithmetic operators (+, -, *, /, &, |) along with the comparison operators (==, ~=, <, <=, >, >=) and a special operator (@) for creating 2D and 3D vectors. These operators can be thought of as patterns that are configured using a special syntax. The relationship defined by these operators depends on the entities involved. For instance, the + operator has different meanings for various entities. For contribution functions, this operator creates another contribution function that, when evaluated by the ViB engine, adds the output of the two input contribution functions. However, for other entities, such as strings and lists, this operator acts as a means to create or combine lists¹.

In our example, the results of the expansions of the channelOutput and delta configurations are combined using the * operator. We presented the expansion of the channelOutput configuration in the previous section. The delta pattern is another primitive that creates a contribution function entity representing the change in an

¹The relationship representing the concatenation of multiple strings to create a single string is provided by the Concatenate pattern.

input function from the previous interval. For example, $\text{delta}(f)$ expands to $f_t - f_{t-1}$ which we denote Δf_t . Therefore, $\text{delta}(t)$ expands to the contribution function denoted Δt which outputs the time between evaluation intervals. The combination of these two expansions results in the creation of the contribution function $\text{velocity}_{t-1}\Delta t$. It is important to note that the $*$ operator itself does not perform a computation but instead composes the two operand entities to form another entity that, when evaluated by the ViB engine, will perform the desired computation.

Node Style Configurations

Figure 5.2 shows the expansion of our ChannelContribution configuration. Since the ChannelContribution configuration is a node style configuration, the bindings made in the resolution frame correspond to those in the configuration. The contribution variable is bound to the results of the combination of the two smaller scale configurations describe in the previous sections.

Figure 5.2: ChannelContribution Expansion

Once the variables are bound, ChannelContribution is expanded to the ViBChannelContribution primitive. This primitive references the channel and contribution variables in order to create the following channel contribution entity.

$$location_t \stackrel{+}{=} velocity_{t-1}\Delta t$$

where $location_t$ represents the affected channel, $velocity_{t-1}$ represents the output of the velocity channel at time $t-1$ and Δt represents the time change between intervals. In essence, this channel contribution entity represents the change in location of a body caused by the body's velocity. In the next section we will see how this channel contribution can be combined with others in order to combine the effects of velocity with those of acceleration as well.

ViB Model Specification

A ViB model is constructed by combining multiple channel contribution entities, some of which may be to the same channel. The combination of multiple channel contributions to form a ViB model is performed using the ViBModel pattern presented in Section 5.1.2. Figure 5.3 shows an example pattern configuration that creates a ViB model entity from a set of channel contributions. The “name” variable is bound to the string “Ball” which is used to name the resulting ViB model entity while the “contributions” variable is bound to a list of channel contribution entities that result from the configuration of the four ChannelContribution patterns.

```
ViBModel{
  name = "Ball"
  contributions = [
    ChannelContribution{
      channel = ["location"]
      contribution = initial(2.0 @ 1.0 @ 0.0)}
    ChannelContribution{
      channel = ["location"]
      contribution = channelOutput(["velocity", -1]) * delta(t)}
    ChannelContribution{
      channel = ["location"]
      contribution = channelOutput(["acceleration", 0] * square(delta(t)) / 2}
    ChannelContribution{
      channel = ["velocity"]
      contribution = channelOutput(["acceleration", 0] * delta(t))}]}
```

Figure 5.3: Simple ViB Model for Kinematic Displacement

The four ChannelContribution configurations in Figure 5.3 result in the creation of corresponding channel contribution entities. In this example, there are three contributions to the channel “location” and one to the channel “velocity”. The first is a contribution that sets the initial output of the location channel at time $t = 0$. The next two contribute the change in displacement to the location channel due to the velocity and the acceleration respectively which are provided using the output of corresponding channels. The final ChannelContribution contributes the change in velocity due to the acceleration to the velocity channel.

The four channel contributions are provided to the ViBModel pattern through the “contributions” variable. Due to the definition of ViBModel presented in Section 5.1.2, ViBModel expands to the ViBModelEntity primitive which creates a ViB model entity from the provided contributions in which multiple contributions to the same channel are blended to form a single channel signal. Hence, the combination of these four channel contributions by the ViBModelEntity primitive results in the following ViB model representation.

$$\begin{aligned} location_0 &= (0 \ 1 \ 0) \\ location_t &\pm velocity_{t-1}\Delta t + acceleration_t\Delta t^2/2 \\ velocity_t &\pm acceleration_t\Delta t \end{aligned}$$

where $velocity_{t-1}$ is the output of the velocity channel on the previous time-step while $acceleration_t$ is the output of the acceleration for the current time-step. This implies that the location and velocity channels are dependant on the current output of the acceleration channel and, therefore, the acceleration channel must be evaluated before the location and velocity channels on each tick. However, there is no ordering required between the location and velocity channels as the location is determined using the velocity from the previous tick. The ViB engine manages channel evaluation ordering, as described in Section 3.4.2.

The configuration in Figure 5.3 does not make any contributions to the acceleration channel and does not set the initial value of the velocity channel. By default, the initial output of a channel is 0. Hence, in the absence of any acceleration, no motion

results.

5.1.4 Pattern Definition

The specification in Figure 5.3 resolves to a ViB model that determines the location of a body kinematically from a supplied acceleration. Kinematic displacement is a general relationship that could be applied in different situations to determine the change in a channel's output using an acceleration instead of a contribution function. So, it is beneficial to capture this relationship as an additional pattern. A *pattern definition* is used to capture common relationships between entities and other patterns. Figure 5.4 shows a pattern definition that allows the output of a channel to be determined using an acceleration. The pattern `accelerationToDisplacement` (to be discussed shortly) provides a displacement contribution of $\frac{1}{2}at^2$.

```
KinematicDisplacementChannel = Pattern [
  ChannelContribution{
    contribution = delta(channelOutput(channel))}
  ChannelContribution{
    contribution = accelerationToDisplacement(acceleration)}}]
```

Figure 5.4: KinematicDisplacementChannel Definition

Although this will produce a similar model to the one presented in the previous section, instead of using a separate velocity channel to simulate inertia, the previous change in the channel is used instead resulting in the following model.

$$channel_t \stackrel{\pm}{=} \Delta channel_{t-1} + acceleration(t)\Delta t^2/2$$

where *channel* is the name of the channel provided to the definition and the *acceleration* component is the result of resolving “`accelerationToDisplacement(acceleration)`”. This amounts to the same calculation but does not require an additional channel for the velocity and is more efficiently evaluated.

Variable Scoping

```

ViBModel{
  name = "Ball"
  contributions = [
    ChannelContribution{
      channel = ["location"]
      contribution = initial(2 @ 1 @ 0)}
    KinematicDisplacementChannel{
      channel = ["location"]
      acceleration = channelOutput(["acceleration"], 0)}}}

```

Figure 5.5: Using the KinematicDisplacementChannel Definition

As previously stated, a pattern definition does not explicitly declare what variables it references. Instead, the input variables of a pattern are implicit in the body of the definition and any smaller-scale patterns used in the definition. To illustrate this, consider the example configuration in Figure 5.5 that produces an equivalent model to the configuration in Figure 5.3 but uses the KinematicDisplacementChannel definition.

The configuration of the KinematicDisplacementChannel pattern binds entities to the channel and acceleration variables. The entities bound to these variables are then accessible in the body of the KinematicDisplacementChannel definition. The ChannelContribution configurations in the KinematicDisplacementChannel definition of Figure 5.4 do not explicitly bind a value to the channel variable but instead inherit the binding of this variable from the configuration of the containing definition. This is shown in Figure 5.6.

Intuitively, KinematicDisplacementChannel expands to two ChannelContributions, each expanding to primitive ViBChannelContributions. Each ViBChannelContribution references a variable named “contribution” in their immediate parents (there are two different “contribution” variables). On the other hand, they also both reference a variable called “channel” which is not defined in their immediate parents but instead is defined one more level up in the expansion chain (where it is common to both).

Figure 5.6: KinematicDisplacementChannel Configuration Resolution

To describe this in more detail, the resolution of the `KinematicDisplacementChannel` configuration results in the creation of a resolution frame containing the bindings for the channel and acceleration variables (frame 1 in Figure 5.6). The expansion of `KinematicDisplacementChannel` then leads to the resolution of two `ChannelContribution` configurations. The resolution of the first `ChannelContribution` configuration results in the creation of a frame containing a binding for the contribution variable (frame 2 in Figure 5.6). `ChannelContribution` expands to the `ViBChannelContribution` primitive which is then resolved. This primitive will then access the contribution variable from frame 2 and the channel variable from frame 1. The same applies to the second `ChannelContribution` configuration but we'll skip the details since they are similar.

This scoping mechanism amounts to dynamic scoping in the pattern resolution

hierarchy. The environment used to provide variables to a primitive is not a single frame but rather a chain of frames that starts at the lowest level configuration and proceeds up the pattern resolution hierarchy. We refer to this variable environment as the *resolution context* of a pattern.

Pattern Signature

A pattern definition may have a signature that allows the pattern definition to be configured using the function-style configuration. The `KinematicDisplacementChannel` definition makes use of an `accelerationToDisplacement` pattern that translates an acceleration to a change in displacement.

```
accelerationToDisplacement = Pattern signature(acceleration)
    acceleration * square(delta(t)) / 2
```

The signature of the above pattern definition allows it to be configured using the function style with a single argument representing the acceleration. However, the existence of a pattern signature and the use of a function style configuration does not change the variable scoping rules. Instead, the ordered arguments in a function-style configuration are bound to the corresponding variables of the signature definition in a resolution frame. Therefore, the pattern configuration

```
accelerationToDisplacement(acceleration)
```

is equivalent to

```
accelerationToDisplacement{
    acceleration = acceleration}
```

since whatever is bound to the `acceleration` variable in the resolution context will be bound to the first variable in the pattern signature of the definition. This amounts to a rebinding of the same value to the variable `acceleration`. However, it does not replace the binding higher in the resolution hierarchy but adds a new binding that obscures the inherited one. Thus, a variable binding in a pattern configuration can be thought of as a variable declaration as well as an assignment. Note that this is the same as in traditional programming languages where we might define a function

```

long factorial (long n) {
    if (n < 1) return 1; else return n * factorial(n-1);
}

```

and then call it with a parameter that coincidentally happens to be called `n` too, as in

```

long n = 5;
long answer = factorial(n);

```

In the body of `factorial`, the internal `n` references the value obtained from the external `n` (value 5).

A pattern signature does not restrict what variable references appear in a pattern definition and it does not prevent the pattern from being configured using the node style. The signature's only purpose is to allow the specification of function-style configurations when desired. Also, not all variables that appear in a signature need to be provided in a function style configuration. The values provided in the configuration will be mapped to the corresponding variable name and any additional variables in the signature will be ignored. For instance, the configuration

```
channelOutput(channel)
```

does not provide the time offset of the channel output function. Therefore, the offset variable will not be bound in the resulting frame and the default offset of -1 will be used (This is specified as part of the `ViBChannelOutput` primitive in Section 5.1.2).

5.1.5 Pattern Libraries

A pattern library allows a set of related patterns to be grouped together as shown in Figure 5.7. But additionally, a pattern library acts as a special type of pattern definition that ensures that all the patterns contained in the library are bound in the resolution context when the library is configured. A library can be configured directly as if it were a pattern definition as long as it contains an entry pattern, which is a pattern definition of the same name as the library. For instance, the `KinematicDisplacementChannel` library in Figure 5.7 can be configured as the corresponding definition was in Figure 5.5.

```

Library KinematicDisplacementChannel {

  KinematicDisplacementChannel = Pattern {
    ChannelContribution{
      contribution = delta(channelOutput(channel))}
    ChannelContribution{
      contribution = accelerationToDisplacement(acceleration)}}

  accelerationToDisplacement = Pattern signature(acceleration)
    acceleration * square(delta(t)) / 2
}

```

Figure 5.7: KinematicDisplacementChannel Library

Figure 5.8: Library Resolution

Figure 5.8 shows the resolution hierarchy that results when a library is configured (as described in Section 4.4.5). Just as with a regular pattern definition, the configuration of the library leads to the creation of a resolution frame (frame 1 in Figure 5.8) that contains the bindings corresponding to those of the configuration. However, when the library is then resolved, two steps take place. First, another resolution frame is created (frame 2 in Figure 5.8) which contains the bindings for the pattern definitions contained in the library and this frame is made a child of the original configuration's frame. In this frame, let entity1 and entity2 be place holders for the body of the definitions in the KinematicDisplacementChannel library

shown in Figure 5.7. The second step is the resolution of the pattern definition of the same name as the library using this frame as its resolution context which expands to the body of the `KinematicDisplacementChannel` definition represented by entity1. Thus, all the definitions contained in the library are accessible when the body of the `KinematicDisplacementChannel` definition is resolved.

In some cases, it may be desirable to resolve a pattern in the context provided by a library without configuring the matching pattern definition. For instance, we may want to use the `accelerationToDisplacement` pattern for another purpose. The primitive pattern `Resolve` provides this capability.

```
Resolve{  
  context = KinematicDisplacementChannel  
  pattern = Pattern accelerationToDisplacement(acceleration)}
```

The above configuration results in the creation of a resolution frame containing the definitions from the library and the resolution of the provided pattern in this resolution context.

Figure 5.8 also illustrates that, like references to pattern definitions, references to libraries are still treated as variable references and that libraries that are configured must be bound in a higher-level resolution frame. Therefore, a pattern library can also contain smaller-scale pattern libraries creating a library containment hierarchy. Also, a pattern library can be related to another library through extension which provides a static relationship that allows scoping similar to class-based inheritance in object-oriented programming. The use of these relationships is described in the next section.

5.2 Pattern Organization

Pattern definitions and pattern libraries give us the ability to define and organize patterns. However, in order to use these facilities to effectively build ViB models, we require the ability to build pattern libraries that provide relationships at different levels or layers of abstraction. In this section, we present the facilities provided to organize patterns. Before we present an example, we give a brief explanation of how

our library organization is derived from Alexander's concept of a pattern language.

5.2.1 Pattern Hierarchy

An important aspect of Alexander's patterns is the hierarchical relationship between patterns that form what Alexander calls a pattern language [2]. A pattern language provides a complete set of instructions for design and includes patterns that represent large-scale relationships such as the layout of regions and towns, along with relationships that represent buildings down to the smallest scale patterns that describe the materials used for construction. Included in the relationship described by a pattern is a set of smaller-scale patterns that fill out or complete the pattern along with any larger scale patterns that bring about the need for the pattern. Also, not only do larger-scale patterns bring about certain smaller-scale ones, they also provide the context that determines the exact nature of how these patterns are realized.

Figure 5.9: Pattern Hierarchy

In a similar way, we use patterns to provide the larger-scale relationships representing a complete animation model, relationships representing common animation model entities, such as controllers, actions and behaviors, and those representing the

low-level channels and contribution functions used to construct ViB models. Figure 5.9 shows a portion of the pattern hierarchy used for ViB model creation. The largest-scale patterns in the hierarchy deal with ViB model creation in general. The next level deals with the creation of sub-models such as the behavioral model that provides action-selection and the action model which maps selected actions to the effects of the degrees of freedom (DOFs) that control a virtual character's low-level movements. An action model is composed of several action entities which represent the effects of particular actions such as walking or running. Each of these actions is a specific instance of a more general action entity that controls when the effects of a particular action are active by weighing the effects using an activation level. The effects of an action need to be properly blended with the effects of other active actions using an appropriate blending scheme. At the lowest level, the control and effects of actions translate to functional relationships between the channels in a ViB model.

High-Level Architecture

In order to illustrate how pattern libraries are used to both arrange patterns and support their use to generate models, we will first present the highest-level of pattern library organization. Although our main focus is to use patterns to create ViB models, in order to test these models, we make use of other architectural components to connect to the ViB engine. Each of these architectures provides a different model associated with a virtual character. Figure 5.10 gives a high-level view of the models we use to constitute a virtual character. The character's geometric model representation is derived from VRML. The ViB2VRML model consists of the connections between the ViB model and the VRML geometry that allows the ViB model to animate and receive feedback from the geometry. The Articulated Puppet Engine (APE) architecture is used to connect the VRML and ViB models together using the ViB2VRML model and also acts to coordinate their evaluation. Consequently, an APE model consists of these three sub-models.

Figure 5.10: Virtual Character Architecture

Pattern Primitives

We have integrated the specification of the above mentioned models into our pattern specification environment. This allows us to specify all the models associated with a virtual character in a unified way using patterns. However, the ability to translate a pattern specification into the corresponding model is not integrated into each of the target architectures. Instead, we integrate pattern primitives into the POCL environment for creating and composing the entities that constitute these models. It is these entities that act as the exchange format between the POCL (Pattern Oriented Composition Language) environment and the target architecture.

The pattern primitives for each of the architecture components are contained in a corresponding pattern library. Figure 5.11 shows the containment hierarchy for these libraries. There are certain compositional relationships that are common across all these domains. These include list manipulation as well as pattern and pattern library manipulation. Pattern primitives that provide these relationships are contained in the POCL library which acts as the root library in the library containment hierarchy. The other libraries in Figure 5.11 contain the pattern primitives that are specific to model creation for the corresponding component. The ViB library contains the pattern primitives mentioned in Section 5.1.2 while the VRML library contains primitives that correspond to the nodes and other facilities supported by VRML.

```

Library POCL {
  Resolve = Primitive ...
  Combine = Primitive ...
  ...
Library ViB {
  ChannelContribution = Primitive ...
  channelOutput = Primitive ...
  ViBModel = Primitive ...
  ...}
Library VRML {
  Transform = Primitive ...
  Sphere = Primitive ...
  ...}
Library ViB2VRML {
  ScalarEffector = Primitive ...
  ...}
Library APE {
  ArticulatedFigure = Primitive ...
  ...}}

```

Figure 5.11: Primitive Pattern Libraries

High-Level Abstractions

Each of the libraries in Figure 5.11 can also be viewed as the largest-scale patterns in a pattern hierarchy that determine what smaller-scale patterns for a particular sub-model are available. The ViB library not only contains the pattern primitives representing ViB model entities but also contains the pattern definitions and pattern libraries that provide patterns representing higher-levels of abstraction, such as those shown in Figure 5.9. Because we are concentrating on animation model creation, the VRML and ViB2VRML libraries simply contain the pattern primitives for each of these components. They also provide a few additional pattern definitions to aid specification. The APE library contains pattern definitions and libraries that are related to virtual character creation. These include general character creation patterns as well as specific patterns representing particular aspects of the character such as its geometry, behaviors and connections between the two.

In the sections that follow, we will use the library structure shown in Figure 5.12.


```

Library POCL {
  Library ViB {
    Library KinematicDisplacementChannel {...}
    Library ActionModel {
      Library Action {...}}
    Library BehavioralModel {...}}
  Library VRML {...}
  Library ViB2VRML {...}
  Library APE {
    Library ArticulatedObject {...}
    Library Ship extends ArticulatedObject {...}}

```

Figure 5.12: Example Library Organization

This structure is a high-level library structure which closely resembles the actual structure we have developed, which is presented in the chapters that follow. In the structure shown in Figure 5.12, the ViB library contains the libraries associated with ViB model creation, such as the ActionModel library, while the APE library contains those patterns and libraries associated with character creation as a whole.

Separation of Specification

Alexander's patterns describe architectural relationships in general terms. An architect may incorporate one or more specific instances of a pattern within a particular design where each instance has its own particular input entities and smaller-scale patterns. In order to support the automatic generation of animation models, we capture the particular instances of a pattern as patterns themselves. Therefore, the specification of an animation model can be viewed as two sets of patterns. One set represents the patterns that are general to the animation model creation process. Examples of these include the Action, SetDOF and SmoothActivation patterns from Figure 5.9. The other set of patterns includes those that are specific to a particular animation model being created, such as the Walk and Run patterns. This separation exists at multiple levels within the pattern hierarchy. For instance, the ChannelContribution and channelOutput patterns are general to both BehavioralModel and ActionModel,

while Action and SetDOF are specific to ActionModel.

In order to support this separation, pattern libraries are used to group related patterns together. For instance, in Figure 5.9, ActionModel is a pattern library that contains those pattern definitions related to action model specification. Action itself is also a pattern library that is contained in the ActionModel library and contains the patterns specific to individual actions. At a higher level, the ViB library contains the pattern primitives related to ViB model creation as well as several pattern libraries related to animation model creation while the APE library contains patterns that use the abstractions provided by the ViB library to create particular ViB models. The following sections contain an example which illustrates how pattern libraries support this type of separation.

5.2.2 The ViB Library

In addition to containing the primitive patterns related to ViB model creation, the ViB library contains pattern libraries that represent common abstractions used to build animation models². The ViB library contains the ViB entry pattern, as shown in Figure 5.13, which configures the “model” pattern to create a set of channel contribution entities which is used by the ViBModel primitive to create a ViB model. For this pattern to make sense, it is sufficient to understand that “model” can represent a list of sub-models or even a list of lists of sub-models. This potentially deeply nested list can be flattened to a one-dimensional list using the Combine pattern (a list manipulation primitive).

The ViB library in Figure 5.13 also contains the KinematicDisplacementChannel library described in Section 5.1.5, as well as the ActionModel and BehavioralModel library. The ActionModel library contains an entry pattern which configures a set of actions and also contains the Action library used to create actions. The specifics of the Action and BehavioralModel libraries is not given here since, at this time, we wish to illustrate the high-level relationship between libraries. Chapter 6 provides a more in-depth look at these abstraction layers.

²As we will see in the next chapter, these abstractions are actually contained in a library that extends ViB but for illustration purposes we will include them as part of ViB here.

```

Library ViB {
  ViBModel = Primitive ...
  ...
  ViB = Pattern ViBModel{
    contributions = Combine(model{}})
  ...
  Library KinematicDisplacementChannel {...}
  Library ActionModel {
    ActionModel = Pattern actions {}
    Library Action {
      Action = Pattern ...
      ...}}
  Library BehavioralModel {...}
}

```

Figure 5.13: The ViB Library

The following configuration illustrates one possible way to use the ViB library from Figure 5.13.

```

ViB{
  model = Pattern [
    MotionModel {...}
    ActionModel {actions = Pattern [Action {...} Action {...} ...]}
    BehavioralModel {...}]
}

```

The above configuration supplies the ViB pattern with a “model” pattern consisting of configurations of several sub-models; MotionModel, ActionModel and BehavioralModel. The resolution of the above ViB configuration will expand the “model” pattern in the context provided by the ViB library. Hence, ActionModel and BehavioralModel will expand to the libraries contained in the ViB library of Figure 5.13.

```

ActionModel {actions = Pattern [Action {...} Action {...} ...]}

```

A similar type of abstraction layering occurs with the ActionModel sub-configuration contained in our ViB configuration. The “actions” pattern configures a set of Action patterns, the details of which are not shown. The ActionModel entry pattern shown in Figure 5.13 configures this “actions” pattern in the context of the ActionModel

library, giving the body of this pattern access to the Action definition provided by the ActionModel library.

The dynamic scoping mechanism supported by POCL allows input to patterns to be provided by higher-level configurations. To illustrate this, consider the following alternate configuration of ViB.

Figure 5.14: ViB Library Expansion

In this configuration, the “actions” variable is bound outside the “model” pattern. However, this binding is still available to the configuration of the ActionModel pattern due to the dynamic variable scoping mechanism. This is illustrated in Figure 5.14 where “actions”, when resolved at the bottom of Figure 5.14, expands to the definition provided in frame 1 of Figure 5.14. In a similar fashion, the MotionModel

configuration expands to the pattern provided in frame 1 since one is not provided in the ViB library frame (frame 2 of Figure 5.14).

This example illustrates how the input to a library can be provided by a higher-level configuration. In the sections that follow, we show how higher-level libraries use this ability to create ViB models while separating the specification of a model from the abstractions that are used to build it.

5.2.3 The APE Library

```

Library APE {
  ArticulatedFigure = Primitive APEArticulatedFigure
  APE = Pattern figure{}
  ...
  Library ArticulatedObject {...}
  Library Ship extends ArticulatedObject {...}}

```

Figure 5.15: The APE Library

The APE library corresponds to the APE architecture shown in Figure 5.10. Consequently, the APE library defines a relationship between models created using the other three top-level libraries shown in Figure 5.12. Figure 5.15 shows an outline of the patterns of interest in the APE library.

Like the ViB library, the APE library contains an entry pattern that allows specifications to be provided to the library. It also contains the ArticulatedFigure pattern which uses the APEArticulatedFigure primitive to create an entity specific to the APE architecture that ties together a ViB model, a geometric model represented using a node structure derived from VRML and the connections between them. This entity can be displayed and animated by the APE architecture.

```

APE{
  figure = Pattern ArticulatedFigure{
    name = "Bob"
    connectors = ViB2VRML{...}
    geometry = VRML{...}
    animation = ViB{model = Pattern ...}}
}

```

The above configuration shows how the APE library can be configured to create an APEArticulatedFigure entity which consists of entities from each of the sub-architectures. The expansion of this configuration is shown in Figure 5.16. The top frame in Figure 5.16 represents the root frame provided by POCL which contains the POCL primitives and the libraries representing different architectures, namely ViB, VRML, ViB2VRML and APE. The expansion of the APE configuration proceeds in a similar fashion to that of the ViB configuration presented in the previous section resulting in the ArticulatedFigure sub-configuration being resolved in the context provided by the APE library.

Before ArticulatedFigure is expanded, each of the sub-models must be expanded and the results assigned to the corresponding variables. The ViB sub-configuration at the bottom of Figure 5.16 will expand to the ViB library bound in the top frame. Expansion would then proceed in a similar fashion to that shown in the previous section, resulting in the creation of a ViB model entity. Although the details of the VRML and ViB2VRML libraries is not shown, a similar process would take place for the VRML and ViB2VRML configurations resulting in the creating of the corresponding entities. These entities would then be used by the APEArticulatedFigure primitive to create the APE entity.

The ViB, VRML and ViB2VRML libraries are not contained in the APE library since it is conceivable that libraries representing other architectures may make use of these libraries as well. However, the above example illustrates how patterns in the APE library can still make use of these libraries due to the supported dynamic scoping mechanism. The APE library shown in Figure 5.15 also contains the ArticulatedObject and Ship libraries, which are discussed in the sections that follow.

Figure 5.16: APE Library Expansion

The ArticulatedObject Library

The ArticulatedObject library supports the creation of articulated figures for use by the APE architecture. The purpose of this library is similar to that of an abstract class in object oriented programming in the sense that it contains patterns that provide relationships that are applicable to a wide range of object types but depend on extending child libraries to provide the specifics.

The ArticulatedObject entry pattern in Figure 5.17 configures ArticulatedFigure with sub-configurations of the appropriate libraries for each component type (connectors, geometry and behaviors) and parameterizes each of these libraries by specifying that each of the input “model” patterns be provided by a corresponding library pattern (ConnectionModel, GeometryModel, AnimationModel). This arrangement

```

Library ArticulatedObject {
  ArticulatedObject = Pattern ArticulatedFigure{
    name = objectName
    connectors = Pattern ViB2VRML{
      model = Pattern ConnectionModel {}
    }
    geometry = Pattern VRML{
      model = Pattern GeometryModel {}
    }
    animation = Pattern ViB{
      model = Pattern AnimationModel {}
    }
  }
  AnimationModel = Pattern [MotionModel{} ActionModel{}]
  GeometryModel = Pattern ...
  ConnectionModel = Pattern ...
  MotionModel = Pattern ...
}

```

Figure 5.17: The ArticulatedObject Library

allows child libraries to provide these models in their own way.

The ArticulatedObject library also contains general versions of each of these patterns. For instance, the AnimationModel definition divides the animation model specification into a MotionModel and an ActionModel. The MotionModel and ActionModel patterns represent sub-models of the animation layer and, therefore, can be provided as layers using pattern libraries as well. The ActionModel library is contained in the ViB library, as shown in Figure 5.13. A general MotionModel is provided in the ArticulatedObject library, the details of which are not shown here (see Appendix B for a more detailed look at this pattern).

This arrangement allows a child library to override only certain patterns as needed. For instance, the AnimationModel pattern need only be overridden if additional animation sub-models are needed. A child library can also override the MotionModel pattern in order to affect this particular sub-model. As stated above, the ActionModel component is provided by the ViB library. Child libraries of ArticulatedObject will tailor this sub-model by providing an “actions” pattern which identifies the types of actions associated with the object. In the next section, we will present a potential child library of ArticulatedObject that illustrates how the above mentioned overriding

can be done.

The Ship Library

The ArticulatedObject library provides most of the patterns required to build a ViB model. The Ship library overrides ArticulatedObject in order to provide the specifics for a particular type of articulated object. The outline of the Ship library is shown in Figure 5.18.

```
Library Ship extends ArticulatedObject {
  Ship = Pattern ArticulatedObject{
    actions = Pattern [Thrust{} TurnLeft{} TurnRight{}]}
  MotionModel = Pattern [super.MotionModel{} ...]
  Thrust = Pattern Action ...
  ...
}
```

Figure 5.18: Ship Library

In addition to the fact that the Ship library extends ArticulatedObject, there are two important aspects to the Ship library. The first is that Ship provides a set of actions that are bound to the “actions” variable in the Ship entry pattern. Since the “actions” variable is bound in a higher-level frame, the binding will be available when the ActionModel is resolved (similar to the expansion in Figure 5.14). Each of the actions (Thrust, TurnLeft and TurnRight) is a specialized pattern that configure the Action library contained in the ActionModel library (see Figure 5.13). The details pertaining to this expansion are not shown here but are included in Appendix B.

The second aspect of interest in the Ship library is the fact that the MotionModel defined in Ship overrides the definition provided in ArticulatedObject. However, the MotionModel in Ship will use the inherited MotionModel and add to it because of the use of “super.MotionModel”. Thus, the super keyword provides library inheritance that is similar to that of object oriented programming languages. Again, the details of the MotionModel definitions and their expansion are not shown here but are included

in Appendix B.

5.3 Summary

In this chapter, we introduced the main ViB entities and their corresponding pattern primitives. These include the `channelOutput` pattern for building contribution functions, the `ChannelContribution` pattern for associating a contribution with a channel and the `ViBModel` pattern for creating a ViB model entity from a set of channel contributions. We used these primitives to both reiterate the syntax and semantics of POCL and to illustrate how a simple ViB model can be constructed using these primitives.

In order to prepare for the chapters that follow, we introduced the architecture we use to represent virtual characters. A character is represented by an APE articulated figure which connects a ViB model to a VRML-like geometric model. We have provided libraries for each of these sub-architectures that contain the pattern primitives to support the creation of the required entities. These libraries are contained within the POCL library, which also contains pattern primitives related to general pattern oriented composition.

Finally, we showed a high-level example in order to illustrate how the dynamic scoping mechanism and the library structure of POCL allows us to build libraries representing layers of abstraction. We used library containment to build smaller-scale abstractions and layers within a layer, as illustrated by the `ActionModel` library of the ViB library. This example also illustrated how a specification involving the abstractions provided by a layer can be organized in a separate library and provided to the layer through the layer's entry pattern. The ViB library contained the lower-level model abstractions while the APE library provided specifications that built models using these abstractions.

In this chapter, we did show some simple definitions and libraries involving the ViB primitives and also provided a high-level look at supporting pattern reuse through library extension. In the next chapter, we present the abstractions and abstraction layers we build on top of the ViB primitives to support higher-level animation model

specification. While presenting these abstractions, we will also provide a more detailed look at library containment and extension.

Chapter 6

Animation Model Specification

In Section 2.3, we discussed various abstractions that are commonly used when implementing animation models. In this chapter, we describe how we implemented many of these abstractions for the ViB model using pattern oriented composition and the ViB model pattern primitives. The entities, and corresponding primitive patterns, that constitute a ViB model are the building blocks for constructing animation models. More specifically, the patterns and libraries implemented for the ViB model include lower-level abstractions, such as those that provide weighted contribution blending, as well as the higher-level abstractions that provide action control and action selection mechanisms.

The main goal of this chapter is to describe the abstractions we developed which allow us to subsequently build virtual characters as described in the next chapter. The details in this chapter are meant to convince the reader that patterns can be used to describe facilities as low-level as variables and as high-level as finite state machines and behavior groups. These details also act to illustrate how certain POCL primitives enable us to accomplish this. It is not the goal of this chapter to ensure that each pattern be understood in detail (although it is a goal to provide the reader with the gist of the patterns). Therefore, we have included, in this chapter, the pattern specifications that we feel highlight the important aspects of each of the libraries involved. The complete versions of the libraries can be found in Appendix C.

6.1 Low-Level Abstractions

The low-level abstractions we will present in this section are contained in the AniMoS (Animation Model Specification) library as shown in Figure 6.1. The AniMoS library extends ViB in order to inherit and, in some cases, extend the ViB primitives. These abstractions are typically provided by or hard-coded into the underlying support architectures of virtual character animation systems. However, this can often restrict aspects of the models created for these architectures. We want to avoid any such restrictions by capturing all but the lowest level abstractions, namely channels and contribution functions, in pattern libraries.

```

Library AniMoS extends ViB{
  LocalIdentifier = Pattern ...
  ChannelContribution = Pattern super.ChannelContribution ...
  System = Pattern ...
  ...
  Library ActivatedChannelContribution {...}
  Library WeightedChannelContribution extends ActivatedChannelContribution {...}
  ...
  Library ChannelVariable {...}
  Library ArticulatedVariable extends ChannelVariable {...}
  ...
  Library VariableSystem {...}
  Library ActivationOrdering {...}
  ...}

```

Figure 6.1: Low-level Pattern Libraries

Some of the abstractions presented, such as variable typing, accessing and encapsulation, are well known abstractions that are built directly into most programming environments. Although it may seem strange to capture these as pattern definitions, we have done so to allow for the creation, modification and extension of these abstractions from within the same specification environment in which they are used. It is definitely possible to provide these abstractions using pattern primitives or even to build these abstractions into the specification mechanism but we felt this would restrict the higher-level abstractions that could be created. Also, at the specification

level, we are not concerned with programming behaviors but, instead, with composing ViB models that can be used by the ViB engine to animate virtual characters. The relationship between the above mentioned abstractions and the task of ViB model composition differs from that of programming and we provide pattern libraries that map these abstractions to suitable ViB model representations.

From the standpoint of a high-level designer using patterns to specify a model, how the low-level building blocks are provided is not important as long as they work and are straight forward to use. However, by providing these low-level building blocks as patterns and pattern libraries within the same specification environment, designers have the option to use what is provided, or to extend, modify or even create new low-level abstractions as required.

6.1.1 Encapsulation

When specifying channel contributions and channel output functions, it is helpful to be able to specify the channel identifiers relative to a containing or encapsulating abstraction. Hence, from the standpoint of a ViB model, encapsulation means that a higher-level pattern can specify the name of a containing system and any channels named in the channel contributions or channel output functions specified in lower-level patterns are then stated relative to the system provided by the higher-level pattern. Furthermore, an intermediate level pattern can nest systems by specifying a system relative to a higher-level pattern that then becomes the containing system for lower-level patterns.

At the same time, we do not wish to restrict all the channel references within a particular lower-level pattern to be within the corresponding ViB model system. The ViB model does not place any restrictions on relationships between channels and we wish to be able to exploit this feature when creating animation models. Therefore, we have defined several patterns in the AniMoS library that aid in localizing channel referencing while still supporting inter-system connections between channels.

Channel Identification

The `ChannelContribution` and `channelOutput` patterns defined in Section 5.1.2 and contained in the ViB library expect to be provided with a channel identifier that contains the full path name for the channel. Consequently, the ability to encapsulate channel contributions and channel output references within a particular ViB model system requires both a mechanism to determine the full system path name of a channel given an identifier local to the encapsulating system and a modification to these patterns to use this mechanism.

We provide the required mechanism as a pattern named `LocalIdentifier`. We will see later how the `LocalIdentifier` pattern is implemented but for now, we will assume that the pattern exists and provides the channel identifier consisting of the full path name given a local channel identifier.

```
LocalIdentifier("location")
```

For instance, if the above configuration were resolved within the “Ball” system then the resulting full channel identifier would be [“Ball” “location”].

Assuming the existence of the above `LocalIdentifier` pattern, any ViB patterns that make use of channel identifiers need to be overridden to use this pattern to determine a channel’s full path name. For this reason, the AniMoS library has been created as an extension of the ViB library and, within this library, the `ChannelContribution` and `channelOutput` patterns have been overridden to use `LocalIdentifier` to determine the full channel identifier. The following is the overridden `ChannelContribution` in the AniMoS library.

```
ChannelContribution = Pattern super.ChannelContribution{
  channel = LocalIdentifier(channel)}
```

The `channelOutput` pattern is overridden in a similar manner.

At the top level of any nested systems, the full identifier is just the identifier itself. Therefore, the following `LocalIdentifier` pattern has been added to the AniMoS library.

```
LocalIdentifier = Pattern signature(identifier) identifier
```

The `LocalIdentifier` pattern simply takes an identifier as a parameter and resolves to

that same identifier. However, as we will see in the next section, this pattern will be overridden to append the encapsulating system path name to the identifier so that this pattern can be used within a system nesting to determine the full channel identifier.

System Encapsulation

By themselves, the patterns presented in the previous section do not change how channel contributions and channel output references occur. However, the addition of the `LocalIdentifier` pattern allows for changes to channel identifier determination without further modifications to the `ChannelContribution` or `channelOutput` patterns. As a result, the `System` pattern that provides encapsulation of channel identifiers need only override the `LocalIdentifier` pattern in order to affect channel identifier determination.

```
System = Pattern children{
  LocalIdentifier = Pattern outer.LocalIdentifier{
    identifier = outer.systemName + identifier}}
```

The `System` pattern has been defined in the AniMoS library in such a manner that all a user needs to do to provide the context for a pattern is to specify the variable “children” as a pattern. This way, it can be used by higher-level abstractions to provide localized channel identification. It accomplishes this by dynamically overriding the `LocalIdentifier` pattern so that any configurations of the `ChannelContribution` or `channelOutput` patterns that occur within the “children” of the system will append the `systemName` to the channel identifiers. The outer qualifier is required in both the `LocalIdentifier` and `systemName` variable references to allow nesting of systems. This is illustrated by the example in the next section.

Example

Consider the following example configuration of the `System` pattern.


```

System{
  systemName = "Ball"
  children = Pattern ChannelContribution{
    channel = ["location"]
    contribution = delta(channelOutput ([["location"]]))}
}

```

The children variable in the above configuration of the System pattern is bound to a pattern in order to allow the System pattern to affect how channel identifiers are determined for any channels identified in the children. Figure 6.2 shows several steps of the expansion of the above example in order to illustrate that identifier ["location"] expands to ["Ball" "location"].

The key to the expansion of the System pattern, for those who wish to follow through the details, is identifier determination. The outer references in the binding for LocalIdentifier and systemName in frame 3 of Figure 6.2 refer to frame 2. Therefore the systemName will be retrieved from this frame and the overridden LocalIdentifier pattern will be retrieved from frame 1 (since frame 2 does not containing a binding for LocalIdentifier) as provided by the AniMoS library. This is important since the children pattern could contain other configurations of the System pattern which override the systemName and LocalIdentifier bindings. Hence, if the "outer" qualifier was not used, Systems could not be nested.

Given that the resulting identifier is ["Ball" "location"], our System configuration expands to the following channel contribution entity.

$$Ball.location_t \stackrel{\pm}{=} \Delta Ball.location_{t-1}$$

Related Patterns

Additional patterns have been defined in the AniMoS library to allow access and modification to channel identifiers. The IdentifierQualifier pattern resolves to the current identifier path while the LocalizeIdentifiers pattern resets the LocalIdentifier pattern so any children of the pattern ignore any higher-level System nesting.

Figure 6.2: Expansion of System Configuration

6.1.2 Channel Contribution Schemes

The ChannelContribution pattern presented in Section 5.1.2 simply defines a relationship between a channel identifier and a contribution function. In some cases, it is necessary to associate an activation level or a weight with the contributions made to a set of channels. In these situations, one or more higher-level pattern specify a channel contribution scheme and an activation level or weight to be associated with any contributions made by lower-level patterns. Then, any channel contributions made by lower-level patterns will be adjusted appropriately using the provided channel contribution scheme and corresponding activation or weight.

In this section, we present several channel contribution schemes, implemented as pattern libraries, that refine the ChannelContribution pattern to include different activation and weighting styles. The implementation of these libraries uses library extension in order to reuse commonality between the schemes.

Activated Channel Contribution

In a ViB model, the output of a channel is determined by blending the output of the contributions from all sources that affect the channel. Some sources may always have an effect on a channel. However, in general, a contributing source only has an effect for a period of time. For instance, a “move forward” behavior will only affect the channels that control motion while the behavior is active. Therefore, the basic channel contribution scheme associates an activation, α_j , with each contribution, f_j , made from source j to channel c such that

$$c_t \stackrel{+}{=} \sum_j f_j(t) \cdot \alpha_j(t)$$

This scheme is implemented in the ActivatedChannelContribution library which is contained in the AniMoS library. The ActivatedChannelContribution library acts as an abstraction layer as described in Section 4.4.5. A higher-level pattern will indicate that lower-level patterns are to use a particular contribution scheme by providing a “children” pattern to the ActivatedChannelContribution entry pattern.

```
ActivatedChannelContribution = Pattern children{}
```

Some lower-level patterns within the “children” pattern may directly or indirectly expand to one or more ChannelContribution configurations.

The ActivatedChannelContribution library overrides the ChannelContribution pattern as follows.

```
ChannelContribution = Pattern [  
  outer.ChannelContribution{  
    contribution = adjustContribution(contribution)}  
  ContributeToActivation{}]
```

In essence, this pattern defines a relationship between the overridden ChannelContribution pattern and an adjustment to the contribution performed by the adjustContribution pattern. The exact nature of this adjustment depends on the definition of adjustContribution. We will see below how ActivatedChannelContribution and the libraries that extend it define this pattern to adjust the contribution appropriately. The use of the outer reference qualifier for ChannelContribution indicates that the overridden ChannelContribution is determined from the resolution context hierarchy and not the static library hierarchy. This is similar to the use of outer for the LocalIdentifier pattern in Section 6.1.1 in that it allows nesting of ChannelContribution definitions

The above ChannelContribution pattern makes use of the smaller-scale patterns, adjustContribution and ContributeToActivation, that are part of the ActivatedChannelContribution library. Use of smaller scale patterns allows extending libraries to override selected relationships within the weighting process while reusing others. The adjustContribution pattern, shown below, weighs the contribution by the activation level via the “weigh” primitive pattern which optimizes the multiplication of the contribution by the activation such that the contribution function is only evaluated when the activation is not zero:

```
adjustContribution = Pattern signature(contribution)  
  weigh(contribution, activation)
```

Due to dynamic variable scoping, the reference of the variable named activation in

the above definition implies that the higher-level patterns that use the `ActivatedChannelContribution` library must supply an activation as well as the channel and contribution used by `ChannelContribution`.

The `ContributeToActivation` pattern, whose details are not shown, creates a separate activation channel in order to keep track of the maximum activation of all contributions made to the channel. The maximum activation of a channel is used by the `ChannelContributionWhenInactive` pattern to contribute a default value to the channel to be used when the maximum activation is zero.

Given the above mentioned patterns, the use of this channel contribution scheme for a particular channel, c , results in the channel signals

$$c_t \stackrel{+}{=} \left(\sum_j f_j(t) \cdot \alpha_j(t) \right) + (f_d(t) \cdot (1 - c.\text{activation}_t))$$

$$c.\text{activation}_t = \max_j \alpha_j(t)$$

where f_j and α_j are the contribution and activation from source j respectively and f_d is the default contribution.

Weighted Channel Contribution

The `WeightedChannelContribution` library is an extension of the `ActivatedChannelContribution` library. Its sole purpose is to separate the activation associated with a contribution into two components, an activation and a weight. The weight is used to adjust the contribution made to the channel while the activation is used to affect the overall activation of the channel as described in the previous section.

```
ChannelContribution = Pattern super.ChannelContribution{
  activation = activation:=(weight)}
```

The overriding `ChannelContribution` pattern uses the provided weight as the activation unless an activation is also provided. The weighting libraries presented in the sections that follow will modify the weights of contributions relative to the weights of other contributions made to a channel while leaving the activation unmodified in order to get the desired contribution blending.

The `adjustContribution` pattern is overridden so that the contribution is adjusted using the weight instead of the activation.

`adjustContribution = Pattern signature(contribution) weigh(contribution, weight)`

The determination of the maximum activation and default contribution remain the same, except that the contributions are adjusted by the weight instead of the activation, giving the following channel signals.

$$c_t \stackrel{\pm}{=} \sum_j (f_j(t) \cdot \omega_j(t)) + (f_d(t) \cdot (1 - c.\text{activation}_t))$$

$$c.\text{activation}_t = \max_j \alpha_j(t)$$

where f_j , α_j and ω_j are the contribution, activation and weight from source j respectively and f_d is the default contribution. By default, the activation and weight are the same but the schemes that follow will change the weight.

Normalized Channel Contribution

In the `WeightedChannelContribution` library, the contributions from each source are weighed independently. This may be acceptable in some situations, such as when higher level entities exist to ensure that only one of the contributions to a channel is active at a given time or when the channel represents a quantity that is amicable to accumulation. However, in some cases, the weight of a contribution may be dependant on the weight of other contributions. For instance, contributions to a channel representing a joint angle may provide the desired angle for that joint. If accumulation of multiple independently weighted contributions is allowed, undesired overcompensation or oscillation in the channel output may occur.

In order to avoid this, the weights from each contributing source can be normalized by adjusting each weight by the ratio of the source's weight to the sum of the weights of all contributions to the channel.

$$\begin{aligned}\omega'_j(t) &= \omega_j(t) \cdot \mu_j(t) \\ \mu_j(t) &= \begin{cases} 0 & \text{if } \sum_i \omega_i(t) = 0 \\ \omega_j(t) / \sum_i \omega_i(t) & \text{otherwise} \end{cases}\end{aligned}$$

where i and j are the set of all contributing sources and μ_j is the scale factor to be applied to the weight from source j . If the weights used for all contributions are continuous, then the normalized weights will be as well.

The `NormalizedChannelContribution` library extends the `WeightedChannelContribution` library and overrides the `ChannelContribution` pattern in order to accumulate the total weight of the contributions to each channel and to adjust the weight applied to each contribution.

```
ChannelContribution = Pattern [
  super.ChannelContribution{
    activation = activation:=(weight)
    weight = weigh(weight, weightScaleFactor{})
  }
  ContributeToTotalWeight{}
```

The `ChannelContribution` pattern uses the original weight as the activation for the overridden `ChannelContribution` and adjusts the weight using the `weightScaleFactor` pattern. The `weightScaleFactor` pattern simply divides the weight by the total weight but uses the `ZeroOnZeroDivide` patterns provided in the `ViB` library to output a 0 for the weight if the total weight of all contributions is 0.

```
weightScaleFactor = Pattern
  ZeroOnZeroDivide(weight, channelOutput(totalWeightChannelIdentifier{ }, 0))
```

The total weight is accumulated much the same way as the maximum activation. Application of this library results in the following channel signals.

$$\begin{aligned}c_t &\stackrel{\pm}{=} \sum_j (f_j(t) \cdot \omega_j(t) \cdot (\omega_j(t) / c.weight_t)) + (f_d(t) \cdot (1 - c.activation_t)) \\ c.activation_t &= \max_j \alpha_j(t) \\ c.weight_t &= \sum_j \omega_j(t)\end{aligned}$$

Strength and Importance Weight Adjustment

It is often desirable to allow a contributing source to have more influence over the resulting behavior than others. To accomplish this, each contributing source can be given a strength that is used to determine its influence relative to other sources. The strength should not affect the weight associated with the contribution but instead should only affect the proportion of a contribution relative to other contributors. In other words, in the absence of other contributors, the strength should not have an effect on the contribution.

This type of blending is accomplished by adjusting the weight scale factor μ_j of a contribution j such that the weight contribution to μ_j from a source i is weighed by the strength of the source τ_i .

$$\mu'_j(t) = \omega_j(t) \cdot \tau_j(t) / \sum_i (\omega_i(t) \cdot \tau_i(t))$$

where $\tau > 0$.

This is accomplished in the `StrengthAdjustedChannelContribution` library by overriding the `weightScaleFactor` and `ContributeToTotalWeight` patterns of `NormalizedChannelContribution` library. Each pattern is overridden to adjust the weight used by the strength of the contribution.

```
weightScaleFactor = Pattern super.weightScaleFactor{
    weight = weigh(strength, weight)}

ContributeToTotalWeight = Pattern super.ContributeToTotalWeight{
    weight = weigh(strength, weight)}
```

Within a single contributing source, some channel contributions are more important than others in realizing the desired behavior. Therefore, an importance can be used to adjust the strength of less important channels.

$$\tau'_i(t) = \tau_i(t) \cdot \iota_{ci}(t)$$

where ι_{ci} is the importance of channel c to source i and $0 < \iota_{ci} \leq 1$.

This is accomplished by overriding the `ChannelContribution` pattern to adjust the strength by the importance.

```
ChannelContribution = Pattern super.ChannelContribution{
    strength = weigh(strength, importance)}
```

Example

In this section, we present, for those interested in the details, an example expansion that illustrates how the contribution scheme libraries presented above adjust a contribution using a supplied weight. Consider the following configuration.

```
NormalizedChannelContribution{
    weight = w(t)
    children = Pattern ChannelContribution{
        channel = "angle"
        contribution = sin(t)}}}
```

Figure 6.3 illustrates the initial expansions caused by the above configuration. First, the frame corresponding to the configuration is created (frame 1 in Figure 6.3) followed by the expansion of `NormalizedChannelContribution`. Since `NormalizedChannelContribution` is an extension of another library, frames for all libraries in the extension chain are created (frames 2,3 and 4 of Figure 6.3). Each of these frames shows only the subset of the actual bindings which we will use to illustrate how channel contribution schemes modify the supplied contribution. Note that the weight used in our example is provided as an unspecified function, `w`, which outputs a value between 0 and 1 when evaluated by the ViB engine.

In Figure 6.4, which is the continuation of Figure 6.3, we focus on the `ChannelContribution` itself and ignore the determination of the maximum activation and maximum weight channels. Each successive configuration of `ChannelContribution` expands to the definition provided by the parent library with the appropriate frames created for each step of the expansion. The last expansion of the outer qualified `ChannelContribution` will begin in the top frame in Figure 6.3. Before this occurs, the `adjustContribution` configuration is expanded using the definition from the `WeightedChannelContributionLibrary` (frame 3 in Figure 6.3) giving the following channel contribution.

Figure 6.3: Expansion involving NormalizedChannelContribution

$$angle_t \stackrel{\pm}{=} \sin(t) \cdot w(t) \cdot (w(t)/angle.weight_t))$$

6.1.3 Variables

One of the main features of the ViB model is that the current output of a channel is determined as a function of its previous output and a set of contribution functions. The exact nature of this function is determined by certain aspects or characteristics of the channel, such as the blend and delta operators. The contribution schemes presented in the previous section can also be used to affect channel output determination. Consequently, patterns that are contributing to a channel must be aware of

Figure 6.4: Expansion of ChannelContribution

the contribution scheme and operators that are associated with that channel in order to do so properly.

The purpose or meaning of a channel dictates which characteristics the channel should have. For instance, a channel representing a joint angle may use the REPLACE delta operator and NormalizedChannelContribution to perform weighted blending of contributions representing absolute joint angles instead of delta values. Each intended purpose has its own set of appropriate channel characteristics. A mechanism for higher-level patterns to capture a channel's purpose is required in order to communicate the purpose to any lower-level patterns that affect the channel.

In essence, a channel is a form of articulated variable. Just as many programming languages associate a type with a variable, we associate a type with certain channels that serve particular purposes. However, in our case, the type of an articulated variable does not identify the type of data associated with the variable but instead identifies the channel characteristics that control channel output determination.

Traditionally, variable declaration and typing is provided as part of the syntax of the specification language. However, our specification mechanism is separate from

our representation. The variables in our specification mechanism, which we will refer to as POCL variables, are used to provide input to patterns. The variables described above, which we will refer to as ViB variables, define a relationship between a channel and the contributions made to that channel.

We provide a set of libraries that represent different types of ViB variables. Using these libraries, higher-level patterns can define and initialize a set of ViB variables to be used by lower-level patterns. Lower-level patterns can then access and modify these variables properly using the variable definitions supplied by the higher-level patterns.

The ChannelVariable library is the root of the ViB variable hierarchy and acts to define the variable protocol which includes patterns for variable definition, initialization, accessing and modification. The other variable libraries inherit from ChannelVariable in order to provide different types of variables. For instance, the ArticulatedVariable extends ChannelVariable in order to associate a channel contribution scheme with a variable.

Channel Variable

The ChannelVariable library defines a relationship between a variable name that identifies the underlying channel representing the variable and a set of definition, initialization, accessing and modifying patterns that act as a protocol for variable manipulation. The patterns in the ChannelVariable library dealing with variable definition and manipulation are outlined in Figure 6.5.

A ChannelVariable is defined using the ChannelVariable entry pattern.

```
ChannelVariable = Pattern Closure{
  context = VariableDefinitionContext{}
  pattern = Pattern Resolve{
    context = variableType{}
    pattern = operation}
  patternSignature = ["operation"]}
```

This definition makes use of the Closure pattern primitive provided by POCL in order to create a pattern definition. As we will see below, the resulting definition can be used to perform operations on a variable by providing an “operation” pattern consisting of

```

Library ChannelVariable {
  ChannelVariable = Pattern ...
  VariableDefinitionContext = Pattern ...
  ...
  InitializeVariable = Pattern ...
  GetVariable = Pattern ...
  GetVariableActivation = Pattern ...
  SetVariable = Pattern ...
  SetVariableDefault = Pattern ...
  LocalIdentifier = Pattern ...}

```

Figure 6.5: Outline of ChannelVariable

one or more configurations of the variable initializing, accessing or modifying patterns.

The Closure primitive is similar to a closure in Lisp and is used to dynamically create a pattern definition from a provided “context” and “pattern”. The “context” of the Closure contains variables from the resolution context where the Closure pattern is resolved. When the resulting pattern definition is later resolved, the variable bindings in “context” are loaded into the resolution context before the “pattern” is resolved. The “patternSignature” variable, bound to [“operation”] in the above definition, indicates the pattern signature to be associated with the pattern definition created by Closure.

The “context” for the above Closure is provided by the VariableDefinitionContext pattern.

```

VariableDefinitionContext = Pattern Context {
  variableName = variableName
  variableQualifier = IdentifierQualifier{}
  variableType = Pattern ChannelVariable}

```

The result of expanding this pattern is a context in which the right hand sides of each of the variable bindings is resolved and bound to the variable in the resulting context. To illustrate this, consider the following configuration of ChannelVariable.

```

ChannelVariable{variableName = “angle”}

```

This configuration expands to the ChannelVariable definition which in turn expands

the Closure pattern. However, before Closure is resolved, VariableDefinitionContext expands to the following context entity.

```
Context {
  variableName = "angle"
  variableQualifier = [...]
  variableType = Pattern ChannelVariable}
```

The variableName in this Context is provided by the above configuration of ChannelVariable. The value bound to variableQualifier depends on where in a System nesting the configuration is located and will consist of a list of strings representing the parent system of the ViB variable¹. As we will see later, variableName and variableQualifier are used to identify the channel representing the ViB variable. The variableType variable is bound to a pattern that indicates the library from which the context was created. This is important as it is the link back to the proper variable operations that make use of the other fields, as we will see below.

As previously stated, the result of expanding the Closure pattern is a pattern definition. In order to use this definition, we must bind a variable to it. For instance, the following creates a pattern definition representing the ViB variable named "angle" and binds it to the POCL variable name "angleVariable".

```
angleVariable = ChannelVariable{variableName = "angle"}
```

Assuming that the above binding was performed in a higher-level configuration, a lower-level one could perform operations on the variable in the following way.

```
angleVariable(Pattern SetVariable{value = 0.5})
```

The expansion of this configuration is shown in Figure 6.6. The initial binding of angleVariable in frame 0 results in the creation of the ViB variable's definition. The configuration of this definition results in the creation of frame 1 containing the "operation" to be performed on the variable. The name of the variable to be bound is determined from the patternSignature provided to the Closure used to create angleVariable. Once this frame is created, angleVariable is expanded which results in the creation of frame 2. Due to the use of Closure in ChannelVariable, the variables in frame 2 of Figure 6.6 were determined by the configuration of ChannelVariable

¹IdentifierQualifier uses LocalIdentifier to determine the encapsulating system path

Figure 6.6: Expansion of a Variable Definition

in frame 0. Hence, `variableName` is bound to "angle". For this example, we have assumed that our `ChannelVariable` configuration is not nested within a `System` so the `variableQualifier` that results from the `IdentifierQualifier` configuration is an empty list (`[]`).

To continue, once the "context" of the `Closure` is loaded, the pattern bound to "pattern" in expanded. This pattern is a configuration of `Resolve` which uses the `variableType` binding from frame 2 in Figure 6.6 to load the appropriate ViB variable library, which in our case is `ChannelVariable`, thus creating frame 3. Finally, the original operation, which is a `SetVariable`, is resolved, using the `SetVariable` from `ChannelVariable` (frame 3).

The `SetVariable` pattern in the `ChannelVariable` library simply performs a `ChannelContribution`.

```
SetVariable = Pattern signature(variableName, value) ChannelContribution{
  channel = variableName
  contribution = value}
```

The expansion of `SetVariable` in Figure 6.6 would use `variableName` from frame 2 and the value specified directly in the `SetVariable` configuration. The other variable manipulation patterns in the library are `InitializeVariable`, `GetVariable`, `GetVariable-Activation`, `SetVariableDefault` and each performs the appropriate operation.

Note that the `LocalIdentifier` pattern shown below is defined in the `ChannelVariable` library. It will override the one presented in Section 6.1.1 and eventually be used in the `ChannelContribution` expansion so that the full variable naming path will be determined using the `variableName` and `variableQualifier` from the context bound in frame 2 of Figure 6.6.

LocalIdentifier = **Pattern** `signature(variableName) variableQualifier + variableName`

Overriding the `LocalIdentifier` in `ChannelVariable` ensures that a variable definition context refers to the proper channel no matter where in a system nesting the variable definition is used.

Articulated Variable

The `ArticulatedVariable` library extends `ChannelVariable` in order to associate a channel contribution scheme with a variable. This involves two main steps. The first is to define a `VariableDefinitionContext` for `ArticulatedVariable` that uses the proper library and contribution scheme for the variable.

```
VariableDefinitionContext = Pattern Context {
  variableName = variableName
  variableQualifier = IdentifierQualifier{ }
  variableType = Pattern ArticulatedVariable
  contributionScheme = contributionScheme := (ActivatedChannelContribution) }
```

In the above definition, the `variableType` uses `ArticulatedVariable` while the `contributionScheme` is bound to the scheme provided or to the default which is `ActivatedChannelContribution`.

The `SetVariable` pattern illustrates how the patterns of the `ArticulatedVariable` library use the ViB variable's channel contribution scheme to contribute the value assigned to the variable to the underlying channel.


```

SetVariable = Pattern signature(variableName, value) contributionScheme{
  children = Pattern ChannelContribution{
    activation = activation(0)
    channel = variableName
    contribution = value}}

```

When the ChannelContribution is resolved in the context of the contributionScheme, the activation is used to weigh the contribution. Unlike a contribution scheme, an articulated variable expects the activation to be a pattern whose signature expects the time interval offset and that resolves to an activation signal. The activation level of the current time interval is used to set the variable. The other variable manipulation patterns in the library are InitializeVariable, GetVariable, GetVariableActivation, SetVariableDefault and each performs the appropriate operation in the context of the contribution scheme.

Other Variable Libraries

The following variable libraries have also been defined.

- The BlendedVariable library provides variables that support the weighted blending of multiple contributions.
- The MemoryVariable library provides variables that are similar to a variable in a traditional programming language in the sense that, once set, a variable retains its value until it is set again. Contribution activations are used to provide this type of setting behavior.
- The ActivationVariable library provides variables that have continuous output between 0 and 1 representing the maximum contribution from multiple activating sources.
- The BehaviorAttribute library provides variables representing internal desire or state levels similar to the internal variables used by Blumberg [11].
- The InhibitableOutput and SuppressibleInput libraries provide variables that allow the inhibition of outputs and suppression of inputs for modules in a subsumption architecture [17].

It should be noted that these libraries inherit the patterns defined by Channel-Variable so they can be used in a similar manner. However, many of them override one or more of these patterns to provide the desired variable characteristics.

6.1.4 Systems with Variables

In Section 6.1.1, we presented the System pattern which encapsulates channel identifiers within a ViB system, and in Section 6.1.3, we presented variables that represented ViB variable typing information. An amalgamation of these two facilities is the ability to encapsulate a set of variables within a system. In this section, we present the VariableSystem abstraction which associates a set of variables with a system, allows access to these variables from within the system using custom pattern names, and allows access to these variables from outside the system while still using the appropriate variable definition. Furthermore, a special EnablingVariable is provided to support the enabling and disabling of VariableSystems in order to improve execution efficiency.

Variable System

The VariableSystem pattern serves two purposes. It encapsulates the channel identifiers of its children in the appropriate system using the System pattern and it relates a set of variables to the children of the system. In order to provide variable support, the VariableSystem pattern initializes the channels representing the variables and resolves its children in a context that provides patterns for accessing and modifying these variables using appropriately named patterns.

The VariableSystem pattern, shown in Figure 6.7, is contained in the VariableSystem library to allow the necessary smaller-scale patterns (some of which will be mentioned below) to be grouped with it. Like the System pattern, it must be provided with a `systemName` (which is available to the System configuration due to dynamic scoping) and a “children” pattern. In addition, it requires a “variables” pattern that resolves to a set of variable contexts created by the variable abstractions presented in Section 6.1.3.

```

VariableSystem = Pattern System{
  children = Pattern [
    InitializeVariables(variables)
    Resolve{
      context = VariableAccessingContext(variables)
      pattern = outer.children}}}

```

Figure 6.7: VariableSystem Pattern

Figure 6.8 shows an example definition that uses a VariableSystem. The children of the VariableSystem configuration map a tempo in cycles per second to a phase between 0 and 2π . Note the use of the custom pattern “SetPhase” for contributing to the phase channel and the use of “tempo()” and “phase()” to access the output of the tempo and phase channels respectively.

```

CycleController = Pattern VariableSystem{
  systemName = "CycleController"
  variables = Pattern [
    BlendedVariable{variableName = "tempo"}
    ChannelVariable{variableName = "phase"}}
  children = Pattern [
    SetChannelToReplaceOutput{
      channel = "phase"}
    SetPhase{
      value = mod(phase() + (delta(t) * tempo(0) * 2 * pi{}), 2 * pi{}})}}

```

Figure 6.8: Example VariableSystem

Variable Initialization

Each of the variables of a VariableSystem must be properly initialized. Each variable library contains the InitializeVariable pattern which performs the required Channel-Contributions to initialize the channels associated with the variable. Consequently,

initialization of the variables is accomplished by applying this operation to each variable using the Collect pattern.

```
InitializeVariables = Pattern Collect(variables{},
    Pattern signature(variable) variable(Pattern InitializeVariable{}))
```

The Collect pattern, named after the Smalltalk collect: method, is a POCL primitive that applies a supplied pattern to a list of entities and expands to the resulting list. The result of supplying the list of “variables” and applying InitializeVariable to each of these is the set of channel contributions that will create and initialize the appropriate channels within the system.

Variable Accessing

The ability to get and set variables using names tailored to the variable is not a necessary part of the VariableSystem abstraction but it does allow for more concise specifications so we have incorporated this into the VariableSystem. The VariableAccessingContext pattern used by VariableSystem, as shown in Figure 6.7, creates a context containing accessing and modifying patterns that have names tailored to each variable. This allows specifications inside the “children” of the VariableSystem to use patterns such as “GetTempo” and “SetPhase” to access and modify the variables “tempo” and “phase”.

Although not necessary for understanding how to use a VariableSystem, we will now present the details of how the VariableAccessingContext pattern creates the context containing the custom pattern names.

```
VariableAccessingContext = Pattern signature(variables) Combine(
    Resolve{
        context = CreateContextArray{
            variables = ["variable"]
            values = [variables {}]}
        pattern = Pattern VariableContext{}})
```

To summarize, the VariableAccessingContext pattern uses the CreateContextArray primitive to create a Context for each variable. The VariableContext pattern is then resolved in each of these contexts to create a corresponding context containing the custom accessing and modifying patterns. The Combine primitive then combines

Figure 6.9: CreateContextArray Resolution

these separate contexts into a single context that is used by the VariableSystem pattern to allow access to the patterns from within the “children”.

To expand the above explanation further, the first step during the resolution of VariableAccessingContext is the creation of the multiple contexts using CreateContextArray. Figure 6.9 shows how CreateContextArray converts a list a variables into a set of contexts in which the variable named “variable” is bound to each of the variables. The Resolve primitive then resolves the VariableContext pattern, shown below, in each of the contexts shown at the bottom of Figure 6.9.

```
VariableContext = Pattern signature(variable)
variable( Pattern CustomizedOperations(variable))
```

The VariableContext pattern applies the CustomizedOperations pattern to a supplied “variable”. The CustomizedOperations pattern (whose details are not shown), is provided in ChannelVariable and the libraries that extend it. This pattern creates a context for a variable in which each operation is bound to a POCL variable derived from the ViB variable’s name. For instance, the pattern bound to SetPhase would be equivalent to the following.

```
SetPhase = Pattern variable (Pattern SetVariable{})
```

where “variable” is the variable provided as input to CustomizedOperations.

Figure 6.10: Combining Variable Contexts

The results of the resolution of the VariableContext pattern for each variable and the combining of these contexts using the Combine primitive is shown in Figure 6.10. This context then acts as the context in which the “children” of the VariableSystem is resolved, thus providing the sub-configurations of this pattern access to the custom patterns.

External Variable Access

In some situations, it is necessary to access or modify the variables of a system from an external source. For instance, as we will see later, a VariableSystem may be used to represent a controller, or some other entity, that has a set of inputs and outputs that act as its interface. In this case, we do not require custom pattern names but we still require a means to ensure that an externally provided variable operation is performed properly for the type of variable involved.

The “variables” for a particular entity are part of the higher-level patterns that use the VariableSystem. This is illustrated in Figure 6.8, where the CycleController pattern provides VariableSystem with the “tempo” and “phase” variable definitions.

In order to ensure that the operations for a particular variable are done properly, we need to extract the proper variable definition from such a higher-level pattern and use this variable definition to perform the desired operation. This capability is provided by the `GetVariableDefinition` pattern.

```
GetVariableDefinition = Pattern signature(variableSystem, desiredVariable) Resolve{
  context = CreateContext{
    variables = VariableNames(variableSystem)
    values = VariableDefinitions(variableSystem)}
  pattern = Pattern ReferenceVariable(desiredVariable)}
```

`GetVariableDefinition` must be supplied with a higher-level pattern that expands to a `VariableSystem` and the name of the desired variable.

```
GetVariableDefinition(CycleController, "tempo")
```

This configuration specifies that the target `variableSystem` is the `CycleController` from Figure 6.8 and the desired variable is “tempo”.

The `GetVariableDefinition` pattern uses the `CreateContext` primitive pattern to create a context in which each variable definition is bound to a POCL variable of the same name, as shown in Figure 6.8. The `ReferenceVariable` pattern, which is a primitive pattern that converts a string to a reference to a POCL variable, is resolved in this context in order to extract the desired variable.

Missing from Figure 6.11 is the expansion of the `VariableDefinitions` and `VariableNames` patterns. It is these patterns that perform the extraction of the variable definitions from the provided `variableSystem`. In actuality, it is the `VariableDefinitions` pattern that performs the extraction and `VariableNames` uses this pattern to then extract just the names of the variables. Hence, we present the `VariableDefinitions` pattern.

```
VariableDefinitions = Pattern signature(variableSystem) Resolve{
  pattern = Pattern ExtractEntity(
    variableSystem{
      VariableSystem = Pattern System{
        children = Pattern TagForExtraction(
          variables{}))}}}
```

We are not going to present the details of the expansion of the `VariableDefinitions` pattern shown above. However, we do wish to point out the use of the POCL

Figure 6.11: CreateContext Resolution

extraction primitives. The key to the extraction is the redefinition of the VariableSystem pattern itself, such that the pattern resolves to only the “variables” portion of the variableSystem. It is this redefinition that allows the extraction of the desired portions (in this case, the variables) of a pattern.

System Enabling

An important aspect of the ViB Engine is the ability to disable subsystems within a ViB model. Once a subsystem is disabled (with a small grace period to allow the children of the system to respond to the disabling), the ViB Engine will no longer evaluate the system or any of its channels or subsystems until the system is enabled again. This allows systems that are not currently participating in the behavior determination process to be pruned from the evaluation process thus improving run-time efficiency.

At the system level of abstraction, we are not concerned with determining when it

is suitable for a system to be disabled. That is left to higher-level abstractions built on top of systems. Instead, we are concerned with providing the ability to enable and disable systems so that higher-level abstractions can make use of them when required.

System enabling is provided through the use of a special variable that is implemented in the `EnablingVariable` library that is an extension of `ChannelVariable`. By including this variable in a `VariableSystem`, that system automatically has the ability to be enabled and disabled. An `EnablingVariable` is always named “enabled” since that is the channel that is interrogated by the ViB engine. It also has an `enablingScheme` which is used to determine how the system containing the variable can be enabled and disabled. On initialization, the “enabled” channel is set to OR all contributions made to the channel (which also sets the delta operator to REPLACE) and the `enablingScheme` is applied.

```
InitializeVariable = Pattern [
  SetChannelToOrContributions{
    channel = ["enabled"]}
  enablingScheme{}
```

Each enabling scheme is provided by an associated pattern defined in `EnablingVariable`. The `EnabledWhileActivated` scheme is used to enable the system when one of the contributions to the channel is active (that is, equal 1). The `EnabledAlways` scheme, ensures that the system is always active by contributing a value of one directly to the channel. The `EnabledUntilDisabled` scheme will ensure that, once enabled, the system will remain enabled until it is explicitly disabled. This is accomplished by creating another channel to receive the disable signal and using this channel to determine if the system should remain enabled.

```
EnabledUntilDisabled = Pattern [
  ChannelContribution{
    channel = ["enabled"]
    contribution = and([ channelOutput(["enabled"]) not(channelOutput(["disable"]))])
  }
  SetChannelToOrContributions{
    channel = ["disable"]}]
```

The `EnablingVariable` library also provides a `Disable` pattern to provide a means to disable the variable’s system.

6.1.5 Other General Pattern Libraries

Gradual Activation

Examples of different types of activation signals are shown in Figure 6.12. It is common in animation models to convert an edge activation to a gradual one, such as the linear and smooth signals in Figure 6.12. This capability is provided by the `LinearTransition` and `SmoothTransition` libraries within the AniMoS library.

Figure 6.12: Types of Activation Signals

`LinearTransition` maps an edge activation signal provided through the “activate” variable to a linear transition between 0 and 1 on activation or between 1 and 0 on deactivation.

```
LinearTransition = Pattern [
  ChannelContribution{
    contribution = if(activate, delta(t) / activationTime, -1 * delta(t) / deactivationTime)}
  ClampChannelOutput{
    range = [0.0, 1.0]}]
```

The `activationTime` and `deactivationTime` variables provide the transition lengths for activation and deactivation respectively. The output of `LinearTransition` is contributed to the channel identified by the provided “channel” variable.

```
LinearTransition{
  channel = ["level"]
  activate = channelOutput(["activate"])
  activationTime = 1.0
  deactivationTime = 0.5 }
```

Due to dynamic variable scoping, the channel variable binding in the above configuration is available to the `ChannelContribution` configuration in the definition of

LinearTransition. Consequently, the output of the above LinearTransition configuration is contributed to the “level” channel.

In some cases, a transition function whose derivative is 0 at the beginning and end of the transition is desired. This can be accomplished using a cubic function with these characteristics over the input range of 0 and 1 [23] combined with a LinearTransition to provide the required input to the function.

```
SmoothTransition = Pattern [
  LinearTransition{
    channel = channel + ["linear"]}
  ChannelContribution{
    contribution = smoothstep(channelOutput(channel + ["linear"], 0))
  SetChannelToReplaceOutput{}}
```

The smoothstep pattern uses the cubic function $3x^2 - 2x^3$ which has the required characteristics.

```
smoothstep = Pattern signature(input) square(input) * (3 - (2 * input))
```

Activation Ordering

Another common arrangement in animation models is a group of mutually exclusive entities. The purpose of such a group is to ensure that only one of its members is active at a given time. The easiest way to provide this type of arrangement within a ViB model is shown in Figure 6.13. In this arrangement, we assume that each entity is contained within its own system. The *activate* channel represents a request for the corresponding system to be activated and is provided by an external source. The *canActivate* channel is used in combination with *activate* to determine if the system is activated.

$$system_i.activated_t = system_i.canActivate_t \wedge system_i.activate_t$$

A system is eligible for activation if no higher priority system is already active. This is determined for the systems in the group by propagating the *canActivate* condition through the priority ordering of systems such that a system can only be

Figure 6.13: Activation Ordering

activated if its immediate predecessor could be ($system_{i-1}.canActivate_t$) but is not ($\neg system_{i-1}.activated_t$).

$$system_i.canActivate_t = system_{i-1}.canActivate_t \wedge \neg system_{i-1}.activated_t$$

The above relationships are provided by the ActivationOrdering library.

```
ActivationOrdering = Pattern [
  DetermineWhenActivated{}
  InjectCanActivate{}
  PropagateCanActivate{}]
```

The ActivationOrdering pattern uses a provided ordered set of systemNames and several smaller-scale patterns to connect the appropriate channels within the named systems in the manner described above. The following ActivationOrdering configuration specifies that the “Walk” system has a higher priority than “Run” such that if both are requested to be active only “Walk” would be activated.

```
ActivationOrdering{
  systemNames = ["Walk" "Run" "Crawl"]
  activate = channelOutput(["allowMotion"]) }
```

In addition to the systemNames, an ActivationOrdering is provided with an edge activation signal through the “activate” variable that is used as the canActivate condition of the first system in the ordering and, hence, allows the entire ordering to be activated or deactivated.

We have not included the details of the smaller-scale patterns used in the ActivationOrdering pattern but we will summarize what they do. For each system, the

DetermineWhenActivated pattern sets both the canActivate and activated channels to AND all contributions and makes the necessary contributions to the activated channel. The InjectCanActivate pattern contributes the edge activation signal provided by the “activate” variable to the canActivate channel of the first system in the ordering while the PropagateCanActivate pattern propagates the activation status through the remaining systems. These patterns make use of POCL primitives for accessing a particular element of a list and for extracting sub-lists from a list.

6.2 Animation Model Abstractions

The pattern libraries presented in the previous section provide the basis for the definition of patterns and libraries representing the abstractions commonly used to build animation models for virtual characters. We make use of these low-level libraries in order to help define pattern libraries representing higher level abstractions, such as actions, controllers and action-selection behaviors, that allow the mapping of models built using these abstractions to the underlying ViB model representation.

Figure 6.14: Animation Model Architecture

Like many architectures for virtual characters [11, 71, 55, 9], we have divided

the overall animation model into several sub-models or layers. Our animation model architecture is shown in Figure 6.14. The layers have well defined interfaces between them which act as abstraction barriers in order to provide independence between how the abstractions of each of the layers is provided. Corresponding to each of the layers in Figure 6.14 is a pattern library that contains the patterns that implement the different abstractions of the layer. The layers are categorized as follows.

- The MotionModel provides low-level motion through the environment and enforces any required constraints that are not enforced in the geometry itself.
- The ControllerModel provides controllers that use state information and controller inputs to determine the value of the controller outputs.
- The ActionModel provides actions representing motor skills that, when activated, act to map controller output values to the appropriate effects on the geometry.
- The BehavioralModel provides the action-selection mechanism that maps sensor input and state information to controller input and action activations.

The Sensor and State layers act to provide information to the models. These models are often character specific so they are not contained in AniMoS but instead are provided by higher-level abstractions, such as those presented in the next chapter.

Figure 6.15 shows the libraries used to provide the common animation model abstractions. In this section, we will describe how we built each of these libraries using the abstraction presented in the previous section. Before we proceed, we would like to point out that, as in Section 6.1, many of the patterns representing animation model abstractions define a relationship between a name, an activation level and a set of effects. The name is used to name the system containing any channels and subs-systems while the activation level is used to trigger the effects. The effects are typically a pattern that uses the provided activation level in some way. The activation level of a pattern may be solely determined by the pattern itself or may be partially or wholly determine by larger-scale patterns.

```

Library AniMoS extends ViB {
  Library ControllerModel {
    Library ActionController {...}
    Library PostureController {...}}
  Library ActionModel {
    Library Action {...}
    Library ActionGroup {...}
    Library ActionLayer {...}}
  Library BehavioralModel {
    Library Module {...}
    Library Behavior {...}
    Library BehaviorGroup {...}}
  Library FiniteStateMachine {...} }

```

Figure 6.15: Higher-Level Pattern Libraries

6.2.1 Motion Model

In our architecture, the Motion Model acts as an abstraction barrier between the geometry and the higher-level action and controller models in order to free these models from dealing with low-level concerns such as collision detection and surface or resting contact between a character and other objects. The use of physics-based models or other types of constraints integrated with the geometry overlap with the facilities associated with the Motion Model. In essence, this layer acts to compensate for the lack of such facilities in the underlying Geometry architecture. Because of the tighter coupling between the MotionModel and the underlying geometric model, the AniMoS library does not provide a general MotionModel library but does provide certain facilities that can be used by this model, such as the KinematicDisplacementLibrary presented in Chapter 5.

Motion

Motion of a body is controlled using pattern libraries that model kinematic displacement. These libraries provides the ability to apply acceleration effects such as gravity, inertia and damping to a channel that represents a location or angle of orientation.

This type of displacement determination works well for free floating bodies but requires the use of constraints, as described in the next section, to handle the effects of contact with other objects.

Constraints

Constraints that deal with low-level motion issues such as collision detection and contact are more suitably dealt with when tightly integrated with the Geometry. However, in those situations when such facilities are not available, it is possible to approximate constraints within a ViB model. We have implemented several different constraint controllers which act to provide collision detection and resting contact for virtual characters. These constraints are specific to the particular architecture that provides the geometric model and are covered in Section 7.6.

Interface

The MotionModel directly controls the global positioning of the character. The exact nature of this control depends on the constraint controllers that are present in the motion model. Constraint controllers can be enabled using the EnableController pattern. However, constraint controllers are only enabled by the PostureController (presented in the next section) and all higher-level models communicate their motion requirements through this single controller. This provides a separation between the low-level motion determination and higher-level behavior layers.

6.2.2 Controller Model

The Controller Model acts as an abstraction barrier for the Action Model by translating state information and controller input into controller output that is used to determine the effects of actions on the Geometry. The Controller Model also provides additional state information representing the current posture of the character's body that can be used to aid higher-level behaviors in action-selection.

In addition to containing libraries for the various abstractions of the ControllerModel layer, this library also contains the ControllerModel pattern which is used by

higher-level patterns to convert a set of controller specifications into the corresponding ViB model representation.

```
ControllerModel = Pattern Resolve{
  context = ActionInterfaceContext{}
  pattern = controllers}
```

In order to use this pattern, the controllers variable should be bound to a pattern containing the specification of one or more ActionControllers and a PostureController. This is similar to the abstraction layer present in Section 5.2.2. We will see how this pattern is typically provided with a set of “controllers” in Chapter 7. The ActionModel and BehavioralModel libraries provide similar patterns for configuring a set of entity specifications.

Action Control

Action controllers map state information and inputs from higher-level behaviors into a form that is usable by the Actions in the Action Model to properly control the DOFs of a Geometry. An ActionController is simply a VariableSystem whose variables are the inputs and outputs of the controller.

```
ActionController = Pattern VariableSystem{
  systemName = controllerName
  variables = Pattern inputs{} + outputs{}
  children = effects}
```

The effects of a controller is a pattern that defines how the input and state information is mapped to the outputs and is typically hand crafted for a particular purpose.

Figure 6.16 shows an example ActionController that translates a desired tempo into a phase value between 0 and 2π that could be used to coordinate cyclic motions performed by a set of flapping wings. Other examples are presented in Section 7.7.

Posture Control

Some actions and behaviors cause the body of the character to be in certain postures while others are only applicable in certain postures. The posture controller provides low level control and feedback of body posture in order to act as an abstraction

```

FlapCycleController = Pattern ActionController{
  controllerName = "FlapCycleController"
  inputs = Pattern [BlendedVariable{variableName = "tempo"}]
  outputs = Pattern [ChannelVariable{variableName = "phase"}]
  effects = Pattern [
    SetChannelToReplaceOutput{
      channel = "phase" }
    SetPhase{
      value = mod(phase() + (delta(t) * tempo(0) * 2 * pi{}), 2 * pi{})}}]

```

Figure 6.16: Example Action Controller

barrier between the entities that determine posture and those that operate in certain postures. In addition to acting as an abstraction barrier, a PostureController also identifies effects that should be active while in a particular posture.

The PostureController pattern defines a relationship between a set of Postures and a set of valid transitions between these postures. Basically, a PostureController is an FSM that defines transitions between postures in terms of state information or requests from higher level entities. Each Posture consists of a name, a set of effects that are active when the posture is active and a set of valid transitions to other postures. An example PostureController is shown in Figure 6.17.

In most cases, transitions are requested from a higher-level entity using the RequestPosture pattern (presented later). For instance, if the controller in Figure 6.17 were in the “Falling” posture and the “Flying” posture was requested (as indicated by wantFlying {}) then the controller would transition to “Flying”. Some transitions depend on state information or a combination of conditions. In Figure 6.17, contact with a surface (as indicated by the surfaceContact{} condition) will cause a transition from “Falling” to “Landed” while no surface contact and a downward velocity will cause a transition from “Landed” to “Falling”.

The effects of a Posture allow the PostureController to compensate for facilities that are lacking in the underlying Geometry. In Figure 6.17, each posture activates an action using ActivateAction which is used to provide the default pose of a character in

```

PostureModel = Pattern PostureController{
  startingPosture = "Falling"
  postures = Pattern [
    Posture{
      postureName = "Falling"
      effects = Pattern ActivateAction(["PostureActions" "Fall"])
      transitions = Pattern [
        Transition(wantFlying{}, "Flying")
        Transition(surfaceContact{}, "Landed")]
    Posture{
      postureName = "Flying"
      effects = Pattern ActivateAction(["PostureActions" "Fly"])
      transitions = Pattern [
        Transition(not(wantFlying{}), "Falling")
        Transition(wantStanding{}, "Landed")]
    Posture{
      postureName = "Landed"
      effects = Pattern ActivateAction(["PostureActions" "Stand"])
      transitions = Pattern [
        Transition(wantFlying{}, "Flying")
        Transition(and([downVelocity{} > 1 not(surfaceContact{})]), "Falling")]]}

```

Figure 6.17: Posture Model for a Flying Character

the absence of any other active actions. For instance, the above mentioned transition from "Falling" to "Flying" would cause the ["PostureActions" "Fall"] action to be deactivated and the ["PostureActions" "Fly"] action to be activated. These actions must be properly incorporated into the ActionModel of the character in order that they provide the default values for the DOFs of the character. How this is accomplished is discussed in the Section 6.2.3. The effects of a posture can also include the activation of constraint controllers to ensure that the proper low-level motion constraints required for a particular posture are enforced. Examples of activating constraint controllers from a PostureController are shown in Section 7.7.

Interface

The interface of the ControllerModel is defined by a set of patterns that are provided by the AniMoS library in order to allow other models to influence the entities specified

for the controller model. The following are the patterns that allow the entities of other models to communicate with controller model entities.

- `RequestPosture(postureName, activation)` signals to the posture controller that the named posture is desired when the provided activation signal is 1.
- `currentPostureIs(postureName)` expands to a contribution function that indicates whether the named posture is the current posture or not.
- `EnableController(controllerName, activation)` enables the named action controller when the provided activation signal is 1.
- `GetControllerVariable(controllerName, variableName)` expands to a contribution function that provides the output of the channel representing the named variable of the named controller.
- `SetControllerVariable(controllerName, variableName, value, activation)` contributes the provided value to the named variable of the named controller. The provided activation signal is used by the ViB variable library that is associated with the variable being set.

The activation signal used by some of the above patterns is usually provided implicitly since most higher-level entities that use these interface patterns have an associated activation level and this is available to these patterns through the dynamic scoping mechanism of POCL.

6.2.3 Action Model

The purpose of the Action Model is to map controller output to changes in the degrees of freedom (DOFs) of the character's lower-level motion model and geometry. Depending on these models, the DOFs may provide kinematic control through positions, velocities or accelerations, or physics based control through DOFs that determine forces and torques. An action entity is used to define the mapping from controller output to time-varying affects on DOF values. This mapping includes a channel contribution scheme that is used to blend the effects of an action with other actions that

control the same DOFs. The ActionModel library contains the patterns and libraries that support the specification and resolution of Actions, ActionGroups for mutual exclusion and ActionLayers for encapsulating the effects of a set of actions.

Action

The Action pattern is defined in such a way as to provide a default means for determining the action's activation level and time input, and to allow these defaults to be overridden if required.

```

Action = Pattern ActivationControl{
  children = Pattern TimeControl{
    children = effects}}

```

ActivationControl and TimeControl are patterns that provide the desired activation and time signals for the “effects” of the action. Different versions of these patterns exist in order to provide signals with the desired behavior. The default ActivationControl scheme, SmoothActivation, uses either of the gradual activation schemes presented in Section 6.1.5 to convert an edge activation signal to a gradual one. Actions that do not require smooth activation can use EdgeActivation which does not modify the original activation signal. The default TimeControl scheme, ActivationIntegral, uses the integral of the activation level, approximated using Euler's method, such that $t_t = t_{t-1} + \alpha_t \Delta t$ where α_t is the activation level of the action. Other TimeControl schemes include the ActivationTime scheme, which uses the time since activation as the time input, or the ExternalTimeSource scheme which does not provide any special time input but assumes the use of the global time signal or controller input.

The Action library provides the SetDOF and SetDOFDelta patterns which are used in the “effects” pattern of an Action to map controller output or time-based functions to the affects on the DOFs as shown in Figure 6.18. SetDOF is used to provide the desired value signal for the DOF and converts it to a delta signal, as required by the ViB model, while SetDOFDelta is used to provided a delta to the DOF. Both these patterns use a channel contribution scheme to determine how the effects are blended with those of other actions. The default blending scheme is NormalizedChannelContribution but this can be modified for a particular action

```

Flap = Pattern Action{
  actionName = "Flap"
  phase = GetControllerVariable("FlapCycleController", "phase")
  activationTime = 0.25
  effects = Pattern [
    SetDOF{
      dofName = ["Left" "Wing" "flexion"]
      value = sin(phase)}
    SetDOF{
      dofName = ["Right" "Wing" "flexion"]
      value = sin(phase)}}}]

```

Figure 6.18: Flap Action

to `WeightedChannelContribution` or any of its extensions presented in Section 6.1.2. The activation level of the action, provided by the `ActivationControl` scheme, is used as the weight of the contributions.

Action Group

An `ActionGroup` uses `ActivationOrdering` to ensure only one of the actions in the group is active at a particular time. In an action group, multiple actions will only be affecting the same DOF during transitions. Therefore, the `WeightedChannelContribution` scheme is sufficient to blend the effects of the actions as long as the effects of the grouped actions are not directly combined with other actions that are not in the group. This is usually the case since `ActionGroups` are used in conjunction with `ActionLayers` to isolate the effects of the actions in the group.

Action Layer

An `ActionLayer` encapsulates the effects of a group of actions within a subsystem using the `System` pattern. The benefits of supporting this type of encapsulation is that certain subsets of actions may require different interaction and blending characteristics. Figure 6.19 shows an `ActionLayer` that organizes the default actions for the

PostureController shown in Figure 6.17.

```

PostureLayer = Pattern [
  ActionLayer{
    layerName = "PostureActions"
    actions = [Fly Fall Stand]
    LayerActivationControl = Pattern OneActive{}}
  ConnectSourceLayerToTargetLayer{
    sourceLayer = "PostureActions"
    targetLayer = []}]

```

Figure 6.19: Action Layer for Posture Control

The LayerActivationControl of an ActionLayer determines how the activation of the actions within the layer are coordinated. If LayerActivationControl is set to OneActive, as in Figure 6.17, then an ActionGroup is used to ensure that only one of the actions in the group is active at a time. With MultipleActive, any number of actions may be active and the effects of the actions are blended normally.

Because the effects of the actions in an ActionLayer are encapsulated in the layer's subsystem, the layer's DOF channels must somehow be associated with the actual DOFs of the Geometry. This can be accomplished in two ways. The first approach, which is used for the posture layer in Figure 6.19, is to use the ConnectSourceLayerToTargetLayer pattern to connect the DOF output of a layer to be the default input of another layer or the Geometric DOFs themselves. This allows layer ordering similar to that proposed by Perlin [54] in which the DOF output of higher level layers, when active, obscure the effects of lower-level ones.

The second approach is to contribute the output of the DOFs of a layer to the DOFs of another layer or directly to the Geometry DOFs as if the layer was an action itself using the ContributeSourceToTargetLayer pattern. When a layer is contributed in this way, the activation and total weight associated with each DOF within the source layer is used as the activation and weight of the contribution to the target layer.

Interface

The interface to the `ActionModel` allows higher-level models to activate actions and query their activation status. It includes the following patterns.

- `ActivateAction(actionName, activation)` uses the provided edge activation signal to activate and deactivate the named action.
- `actionActivated(actionName)` expands to a function that indicates whether the named action is currently active (activation level is 1).
- `actionDeactivated(actionName)` expands to a function that indicates whether the named action is no longer active and has been smoothly deactivated (activation level is 0).

An Action can make use of `GetControllerOutput` to reference controller output and `EnableController` to ensure that a controller remains enabled while the action is active. An Action affects the channels representing the DOFs of the geometry using the `SetDOF` and `SetDOFDelta` patterns.

6.2.4 Behavioral Model

The `BehavioralModel` provides the action-selection mechanism for a virtual character. It translates sensor and state information into action activations and appropriate values for controller inputs. Two common action-selection mechanisms are behavior hierarchies and behavior networks. In [11], Blumberg used a behavior hierarchy to form the complete action-selection mechanism for a character. To form such a hierarchy is very difficult. However, creating subsumption behavior networks, such as those proposed by Brooks [17] is also a difficult task. Consequently, we wish to investigate the possibility of combining these mechanism within the behavioral model for a character. We use Modules to encapsulated `BehaviorGroup` sub-hierarchies or `FiniteStateMachines` and connect them together to form behavior networks. We also make use of a small hierarchy of `BehaviorGroups` to determine a character's overall intent which is a similar separation that was used by Tu [71].

Module

Figure 6.20 shows an example using the Module pattern which provides a relationship representing the module entity proposed by Brooks [17]. Module defines a relationship between a set of input, output and internal variables and a set of effects on those variables. The effects of the Module in Figure 6.20 are a FiniteStateMachine but could also be another type of action-selection mechanism, such as a BehaviorGroup. The purpose of a Module is to provide the ability to build encapsulated entities that have an explicit interface and whose inputs and outputs can be connected with other entities in a network or subsumption architecture.

```
FocusOnObject = Pattern Module{
  enablingScheme = Pattern EnabledUntilDisabled{}
  moduleName = "FocusOnObject"
  inputs = ["object"]
  outputs = ["targetLocation" "focus"]
  focus = ActivationVariable
  effects = Pattern FiniteStateMachine{
    initialState = "Start"
    states = Pattern [
      State{
        stateName = "Start"
        effects = []
        transition = Pattern EventDispatchTransition{
          events = Pattern EventTransition(objectArrived{}, "Record")}}
      State{
        stateName = "Record"
        effects = Pattern SetTargetLocation{
          value = externalChannelOutput(object{}, ["location"])}
        transition = Pattern ImmediateTransition("Focus")
      State{
        stateName = "Focus"
        effects = Pattern SetFocus{}
        transition = Pattern EventDispatchTransition{
          events = Pattern [
            EventTransition(objectArrived{}, "Record")
            TimedEventTransition(1, "Start")]]}}}]}
```

Figure 6.20: FocusOnObject Module

The Module pattern makes use of the VariableSystem pattern in order to encapsulate its effects and provide the effects with the appropriate variable interface patterns. By default, inputs use the SuppressibleInput library while outputs use the InhibitableOutput library which provide the capability to suppress and inhibit communication as described in [17]. The Module pattern converts the variable names to the proper variable definitions required by the VariableSystem. Default variable types can be overridden by binding the desired type to a POCL variable of the same name as the variable as shown for the focus variable in Figure 6.20. In addition to the specified variables, Module also adds an EnablingVariable to allow for enabling and disabling of the effects of the Module.

Finite State Machine

Figure 6.20 also contains an example of a FiniteStateMachine. An FSM is an action-selection mechanism in which all possible transitions between behaviors, represented as States, must be explicitly specified. This allows for control over behavior sequencing such that invalid or undesirable transitions can be prevented. However, it also requires the specification of all possible transitions which can be cumbersome to specify if the number of incoming and outgoing transitions for the states is large. In such a case, BehaviorGroups, as discussed in the next section, may be a more appropriate action-selection mechanism.

The FiniteStateMachine pattern defines a relationship between an initial state and a set of states. A State has a name, an activation level, effects and a transition. The name of the State is used to encapsulate the channels related to the activation and transition characteristics of the State in a System. The FSM uses EnablingVariables for each state to control state activation and transition. The effects can contain variable modifications if the FSM is contained within a Module or another type of VariableSystem or smaller-scale action selection mechanisms, such as other FSMs or BehaviorGroups, that are activated when the state is.

The transition of a State determines how and when activation is transferred to another state once the state becomes active. We have implemented different types

of transition types which are similar to those proposed by Brooks [17]. The ImmediateTransition pattern transfers activation to the next state immediately² and is used for states that modify one or more variables. The ConditionalTransition pattern also performs an immediate transition but uses a condition to determine which of two states to activate. Finally, the EventDispatchTransition consists of a set of smaller scale patterns that associate a target state with either a condition or a time-out value and will transfer activation to the state associated with the first of the conditions to be satisfied.

Each state in an FSM reacts to the activation level of a higher-level entity that contains the FSM, such as a Module. The starting or initial state becomes active when the higher-level containing entity is activated. The activation is then passed from one state to another based on the transition models. If the higher-level entity containing the FSM is deactivated, then all states become deactivated and, in essence, the FSM is reset.

Behaviors and Behavior Groups

A BehaviorGroup is an action-selection mechanism that ensures that only one of the behaviors in the group is active at a particular time. The exact mechanism used is described below but, in essence, each Behavior in a group has a value that is a function of a set of scalar dependency signals and the values of the other behaviors in the group. The functional combination of dependencies and behavior values ensures that, within the group, only one behavior has a value that is non-zero and, thus, is active. Although BehaviorGroups do not restrict what transitions between behaviors in the group take place, dependency scaling is important in arriving at the proper action-selection. This is illustrated in the example shown in Figure 6.21 where certain dependencies are scaled in order to arrive at the desired behavior transitions.

The Behaviors in a BehaviorGroup define a relationship between a set of dependencies and a set of effects. As with FiniteStateMachine, the effects of a Behavior can be variable modifications if the BehaviorGroup is contained within a Module or it can be a smaller-scale action selection mechanism. In this way BehaviorGroups can

²By immediately, we mean in the next time-step.

```

ApproachLocation = Pattern BehaviorGroup{
  children = Pattern [
    Behavior{
      behaviorName = "LocationReached"
      dependencies = [atLocation{}]
      effects = Pattern SetArrived{}
    }
    Behavior{
      behaviorName = "LocationChanged"
      dependencies = [not(desiredLocation{} == targetLocation{}) * 10]
      effects = Pattern [SetTargetLocation(desiredLocation{})]
    }
    Behavior{
      behaviorName = "MoveTowardsLocation"
      dependencies = [0.1]
      effects = Pattern [
        BehaviorGroup{
          children = Pattern [
            Behavior{
              behaviorName = "FaceLocation"
              dependencies = [not(facingObject{}) * 10]
              effects = Pattern SetTurn{}
            }
            Behavior{
              behaviorName = "NothingInTheWay"
              dependencies = [0.1]
              effects = Pattern [SetMove{}]}
            Behavior{
              behaviorName = "ObjectInTheWay"
              dependencies = [objectInWay{}]}
            effects = Pattern [{}]}]}]}

```

Figure 6.21: ApproachLocation Behavior Group

be combined hierarchically to form an action-selection hierarchy such as those built by Blumberg [11].

Each behavior has an associated value which is always greater than or equal to 0 and a behavior is active when its value is greater than zero. Once activated, the effects of the behavior are also activated. Our implementation of the Behavior and BehaviorGroup patterns has been derived from Blumberg's work and use his approach to determine the value of a Behavior. In Blumberg's model, each behavior i has an uninhibited value at time t .

$$behavior_i.uninhibited_t = interest_t \cdot f(dependencies)$$

The level of interest $interest_t$ is a value that ranges from 0 to 1 and is used to ensure that an active behavior that is not achieving its goals does not remain active indefinitely. The level of interest is initially 1 and can be set to decrease steadily while the behavior is active. The dependencies are a set of values that are used to determine the value of the behavior. In Blumberg's model, the dependencies are made up of the values of internal variables and what are referred to as releasing mechanisms, which act to give a scalar range to conditions that exist in the environment. In our model, a dependency can be any signal that provides a scalar value but we do make use of internal variable values and releasing mechanisms as well as other state information and allow mixed combination of values as Blumberg did.

In order to ensure that only a single behavior in the group is active, we use the inhibition scheme described by Blumberg. In this scheme, the behavior values of the behaviors in a group are used to mutually inhibit the other behaviors in the following way.

$$behavior_i.value_t = behavior_i.uninhibited_t - \sum_j N_{ji} \times behavior_j.value_{t-1}$$

where N_{ij} is the inhibitory gain of behavior j on behavior i . When $N_{ij} \geq 2$ and $behavior_i.value_t \geq 0$, this scheme will guarantee that one behavior will win out and inhibit the others.

Interface

The BehavioralModel uses the interfaces provided by the ControllerModel and ActionModel in order to control these models. In addition, entities within the BehavioralModel can communicate between each other. The following patterns are provided to communicate with modules from other model entities such as a BehaviorGroup hierarchy.

- **EnableModule(moduleName, activation)** enables the module when the provided activation signal is 1.
- **SetModuleInput(moduleName, inputName, value, activation)** contributes the provided value to the named input of the module.
- **GetModuleOutput(moduleName, outputName)** expands to a contribution function that provides the output of the channel representing the named output of the module.

Modules are interconnected in various ways to form a behavior network. The following patterns provide these connections.

- **ConnectModuleOutput(moduleName, outputName, destinations)** connects the output of a module to the specified destinations. Destinations can be one or more configurations of the patterns that follow.
- **ToModuleInput(moduleName, inputName)** contributes the value and activation of the output being connected (using **ConnectModuleOutput**) to the named input.
- **ToSuppressInput(moduleName, inputName)** contributes the value and activation of the output being connected to the named input while suppressing, or replacing, any normal input received.
- **ToInhibitOutput(moduleName, outputName)** uses the activation of the output being connected to inhibit the named output of the named module. In other words, when the source output is active, the destination output is prevented from being active.

The following example illustrates how to connect the “targetObject” output of “ApproachObject” to the “object” input of “FocusOnObject”.

```
ConnectModuleOutput{
  module = ApproachObject
  outputName = "targetObject"
  destination = Pattern ToModuleInput{
    module = FocusOnObject
    inputName = "object"}}
```

6.3 Summary

In this chapter, we presented several pattern libraries representing the abstractions used to build models capable of controlling a character's motion. The low-level libraries presented provide system encapsulation, weighted contribution blending and mutual exclusion. Built on top of these low-level abstractions are higher-level libraries representing different animation model layers, which were also presented.

6.3.1 Low-Level Abstractions

The purpose of presenting the low-level animation model libraries was three-fold. The first was to convince the reader that we were able to define and implement a low-level variable facility customizable to a degree that has never been possible before. The second purpose was to describe the facilities that each of the libraries provides. The third purpose was to present enough details of their implementation to illustrate some of the techniques used to build these abstractions using POCL. The libraries presented and the implementation techniques they highlight are as follows.

- The System pattern provides channel identifier encapsulation. Its implementation illustrates the use of dynamic pattern overriding using the outer reference qualifier.
- The contribution combination schemes support different methods of weighted blending for multiple contributions made to a single channel. Their implementation illustrate the use of library extension.
- Variable types capture the details of how higher-level entities should affect shared channels properly. Their implementation illustrate the use of the Closure pattern.
- VariableSystem provides channel identifier encapsulation and access to shared variable respectively. Its implementation illustrates the use of the Resolve, CreateContext and CreateContextArray primitives and the extraction of a portion of a pattern specification for use by other patterns.

These lower-level abstractions are the building blocks for our higher-level abstractions. The `VariableSystem` pattern is used by the `Module`, `ActionController` and `PostureController` libraries to identify the input and output variables of specific entities built using these libraries. `GradualActivation` used by `Actions` in conjunction with the contribution combination schemes to provide smooth transitions between actions and the blending of overlapping effects of simultaneously active actions. `ActivationOrdering` used to ensure mutual exclusion in `ActionGroups` and is also used in FSMs to choose one of potentially many possible transitions.

6.3.2 High-Level Abstractions

The high-level abstractions provided by AniMoS which are used to build animation models are divided into layers. These layers include:

- the `ControllerModel` which provides the `ActionController` and `PostureController` libraries used for low-level behavior control
- the `ActionModel` which provides the `Action` library which maps action controller output to the effects on the DOFs of the geometry, the `ActionGroup` library for ensuring mutual exclusion within a set of actions and the `ActionLayer` library for encapsulating the effects of a set of actions
- the `BehavioralModel` which consists of FSMs and `BehaviorGroups` for performing action-selection and `Modules` for encapsulating these selection mechanisms and creating higher-level behavior networks.

We presented a high-level view of these abstractions in order to enable the reader to get a feel for how to create entities using these abstractions. We also presented the interface provided by each layer. In the next chapter, we will illustrate how we used these abstractions to build a ViB model for a virtual character.

Chapter 7

Specifying a Virtual Character

In the previous chapter, we presented the AniMoS pattern library that provides common abstractions used to build animation models. In this chapter, we will look at how we use these libraries to build a virtual character. First, we review the architecture that we built around the ViB Engine in order to build and test virtual characters. Then, we present the animation model we built for our character followed by a summary of our results.

While presenting our character, we reiterate the model layering presented in the previous chapter with respect to the character. For each layer, we present samples of various entities and any organization arrangements used. In some cases, we present sample pattern specifications while in others we may only describe what the patterns do. In most cases, the pattern details are omitted although the patterns involved can be found in Appendix D. We also present several examples of pattern reuse through library extension.

7.1 Supporting Architecture

In addition to the ViB engine, the supporting architecture consists of the components shown in Figure 7.1. As previously described in Section 5.2, the Geometry component provides the geometric representation of the character and any other objects in the environment. The Connections component acts to provide an interface between the

ViB engine and the Geometry in order to allow a ViB model to control the geometry and to obtain geometric state information. The Commands component is provided to allow the user to input binary commands into a ViB model.

Figure 7.1: APE Architecture

These various architectural components are tied together to form the APE system. At system startup, APE triggers the connection of the animation model to the geometry and, during execution, acts to coordinate the rendering of the geometry with the ticking of the ViB engine. APE also is connected to the POCL compiler to allow regeneration of part or all of a character's animation model during debugging.

In APE, the management of the geometry and ViB model of a single character is provided by an *ArticulatedFigure* entity. The APE library described in Section 5.2.3 contains the *ArticulatedFigure* primitive pattern that expands to this entity allowing POCL to be used to create virtual characters. The APE library also contains the *World* primitive pattern which allows the combination of multiple characters and objects into a single world environment.

Our focus has not been to create a sophisticated low-level architecture for providing the geometry and environmental interactions for our characters. Instead, our focus is to provide a minimal architecture for character geometries and to use the ViB engine to provide all possible levels of animation control. We do not claim that this is the best approach for creating realistic, life-like animations. Obviously, the

more facilities that are provided by a lower-level architectural component, the easier animation model creation becomes. However, our desire is to illustrate that the ViB model is capable of controlling the animation of virtual characters at both the kinematic and behavioral levels and, while not capable of true physics-based modeling, is also capable of incorporating physical notions.

Along with illustrating the capabilities of the ViB model, we also wish to illustrate that pattern oriented composition along with the tools that support it can be used to specify virtual characters in such a way as to promote reuse. To help illustrate this, we have strived to provide the ability to specify complete virtual characters using patterns. Again, we do not claim that this is the best approach but in the absence of a more sophisticated lower-level architecture, it allows animation models to be built and tested within a single environment.

At the same time, we do not wish to restrict the use of the models we created to our particular architecture. Therefore, we have aimed to structure the specification of virtual characters in such a way that portions of a character's specification can be easily modified or replaced in order to accommodate other architectural components. We have accomplished this by layering the animation model in such a way that in the presence of a more powerful underlying architecture, only bottom layers of the model need to be changed.

7.2 Character Model

Figure 7.2 shows the virtual room containing common objects such as a fridge, chair and television. We use a character named Robby to interact with these objects in order to illustrate that the ViB model can be used to provide and control different levels of behavior for a virtual character and that such a ViB model can be constructed using the pattern libraries presented in the previous chapter.

Before we look at the specification for Robby, recall the definition of the ArticulatedObject library from Figure 5.17 in Section 5.2.3. The ArticulatedObject pattern is presented again in Figure 7.3 with a slight modification. The AnimationModel is provided to the AniMoS library from the previous chapter instead of the ViB library

Figure 7.2: Robby's Room

in order to allow `AnimationModel` entities to be specified using the `AniMoS` abstractions. We will focus on the `AnimationModel` pattern which provides the specification used to construct the character's `ViB` model.

```
ArticulatedObject = Pattern ArticulatedFigure{
  name = objectName
  connectors = Pattern ViB2VRML{
    model = Pattern ConnectionModel {}
  }
  geometry = Pattern VRML{
    model = Pattern GeometryModel {}
  }
  animation = Pattern AniMoS{
    model = Pattern AnimationModel {}
  }
}
```

Figure 7.3: The ArticulatedObject Pattern

Figure 7.4 shows the relationship between the main libraries involved in creating Robby. The `Character` library of APE extends `ArticulatedObject` and provides the general patterns related to character creation. Included in `Character` is the `AnimationModel` library which overrides the `AnimationModel` of the `ArticulatedObject` library. We have made `AnimationModel` a library in `Character` to allow the grouping together of the pattern definitions related to animation model specification. Due to

library inheritance, this AnimationModel will be referenced by the ArticulatedObject pattern in Figure 7.3 when the Character library is used.

```

Library APE {
  Library ArticulatedObject {
    ArticulatedObject = Pattern ArticulatedFigure {...}
    ...
  }
  Library Character extends ArticulatedObject {
    Character = Pattern ArticulatedObject {}
    Library AnimationModel {
      AnimationModel = Pattern ...
      MotionModel = Pattern ...
      ...
      SurfaceContact = Pattern ...
    }
    ...
  }
  Library Robby extends Character{
    Robby = Pattern Character {...}
    Library AnimationModel extends super.AnimationModel {
      PostureModel = Pattern ...
      ...
    }
  }
}

```

Figure 7.4: APE Pattern Libraries

Figure 7.5 shows the AnimationModel pattern of Character that is provided in the AnimationModel library. This pattern is used to create a ViB model that is the combination of several different layers in the animation model architecture where each layer is represented as a separate pattern. Since all sub-models of the architecture may require access to state information, these patterns are resolved in the context provided by the StateFeedbackContext pattern. The use of this pattern is described in more detail in Section 7.4.

Each of the sub-models in Figure 7.5 creates a different portion of the resulting ViB model. The ControllerModel, ActionModel and BehavioralModel are all provided by the AniMoS library, as described in Section 6.2. The pattern definitions

```

AnimationModel = Pattern Resolve{
  context = StateFeedbackContext{}
  pattern = Pattern [
    StructuralModel{}
    StateFeedbackModel{}
    MotionModel{}
    ControllerModel{}
    ActionModel{}
    BehavioralModel{}
    SensoryModel{}}]

```

Figure 7.5: AnimationModel Pattern Definition

provided by AniMoS for these sub-models reference the actions, controllers and behaviors variables respectively in order to create the corresponding ViB model channel contributions. The BehavioralModel also references the intentions and attributes variables, which is significant for the specification of Robby that we will present below. The contents of these variables are pattern definitions that make use of the abstractions provided by these models as described in Section 6.2. The other sub-models, namely StructuralModel, StateFeedbackModel, MotionModel and SensoryModel, are specific to characters for the APE architecture and are therefore provided in the AnimationModel library of the Character library.

The Robby library extends Character and contains the pattern specifications that are specific to the Robby character. For instance, as shown in Figure 7.4, the SurfaceContact constraint (presented in Section 7.6) is a pattern that can be used by any character so it is contained in the Character library while the PostureModel (presented in Section 7.7) is a controller that is specific to Robby and is therefore part of the Robby library. Robby overrides the AnimationModel library but does not override the AnimationModel entry pattern within this library so the pattern in Figure 7.5 is used through inheritance.

Figure 7.6 shows the pattern definition that was used to generate the animation model for the character Robby. This large-scale pattern contains variables bound to the entities-to be configured by the AnimationModel sub-models mentioned above.

Notice that all the variables are bound to lists or patterns that expand to lists with the appropriate elements plugged in. For instance, the sensedObjects are used by the SensoryModel to create channels within the ViB model to which an external sensor can make sensor information about those types of objects available within the ViB model while the behaviors variable contains the modules used by the AniMoS BehavioralModel that make up Robby's action-sequencing network.

```

Robby = Pattern Character{
  creatureName = "Robby"
  sensedObjects = ["foodSource" "restArea" "sleepArea" "entertainment" ...]
  contactPoints = [{"Left" "Foot" "location"} {"Right" "Foot" "location"} ...]
  constraints = Pattern [SurfaceContact SurfaceFriction ...]
  controllers = Pattern [PostureModel BodyCycleController BodyTargetingController ...]
  actions = Pattern [PostureLayer LocomotionLayer LookingLayer ....]
  behaviors = Pattern [MoveForward Falling FaceLocation ApproachLocation ....]
  intentions = Pattern [IntentionsModel ]
  attributes = Pattern [
    BehaviorAttribute("fatigued", 0.2, 0)
    BehaviorAttribute("hungry", 0.4, 0)
    BehaviorAttribute("sleepy", 0.1, 0)
    ...]}

```

Figure 7.6: Specification for the Character Robby

In the following sections we describe each of the sub-models of our virtual character and some of the patterns used to create them. The definitions of most of the patterns introduced below can be found in Appendix D.

7.3 Structural Model

The control parameters, or degrees of freedom (DOFs), of the geometry of a character are typically joint angles. From the animation model standpoint, the structural model contains information about the structure of the character that is required to create the channels that will be connected to the DOFs of the geometry. The structural model is also used by the ActionModel to create action layers that encapsulate the

effects on DOFs within a subsystem, as presented in Section 7.8. We represent the structural model of a character using a skeletal model consisting of joints and links. This allows us to use the structural model to create simple geometric skeletons for testing such as the one used for Robby and shown in Figure 7.7. This same model is also used to create the connections between the ViB model and the geometry.

Figure 7.7: Robby's Structural Model

7.4 State Feedback Model

The state feedback model creates a set of channels representing geometric state information. This information includes a set of reference directions relative to the character's orientation such as up, forward and left directions, a set of velocities in each of the reference directions and other low-level information about the character's geometry. Figure 7.8 shows an outline of the `StateFeedbackContext` definition. This definition provides a set of variable bindings that allow access to the state information by the other character sub-models. Each of these state channels can be referenced from within the sub-model patterns using the corresponding variable name due to the use of `StateFeedbackContext` with the `Resolve` pattern in Figure 7.5.


```

StateFeedbackContext = Context {
  currentLocation = channelOutput("location")
  totalVelocity = channelOutput(["location" "velocity"], 0)
  forwardDirection = channelOutput(["direction" "forward"], 0)
  ...
  forwardVelocity = channelOutput(["velocity" "forward"], 0)
  ...}

```

Figure 7.8: StateFeedbackContext Definition

7.5 Sensory Model

The SensoryModel of the Character library provides the BehavioralModel with access to information about other objects in the character's environment. A special object sensor provided by APE injects information about each sensed object into the character's ViB model as the output of a corresponding channel. A channel is created within the character's ViB model for each type of object the character has an interest in. This means that a character can only sense a single object for each type but can sense multiple objects. This allows sensed objects of different types to be available simultaneously to the action-selection mechanism.

The injected object information is in the form of another ViB model that contains the information about the object. The Character library provides the following general patterns for accessing information about sensed objects.

- `sensedObject(objectType)` expands to a function that provides the ViB model representing the sensed object of the given type.
- `sensedObjectActivation(objectType)` expands to a function that provides an activation level indicating if an object of the specified type was recently sensed (activation = 1) or not (activation < 1).
- `externalChannelOutput(object, channel)` is a ViB primitive that expands to a function representing the output of the specified channel in the provided object, which is represented as a ViB model.

Several additional accessing patterns have also been provided to access specific pieces of information about a sensed object by using `externalChannelOutput` to access specific channels (e.g. `sensedObjectLocation(object)`).

Certain higher-level behaviors, such as those proposed by Blumberg [11], make use of releasing mechanisms that map sensor information to scalar ranges in order to influence action-selection. We have provided custom named patterns to allow common releasing mechanisms to be accessed within these behaviors. The following patterns provided scalar releasing mechanisms for objects that represent a “rest area” such as a chair or bed.

- `restAreaPresent{}` uses the activation level of the sensed rest area (as described above) to indicate its presence in the environment.
- `nearRestArea{}` increases from 0 to 1 as the character nears the rest area
- `atRestArea{}` increases from 0 to 1 when the character is in close proximity to the rest area.

Each of these can be tailored by providing distance thresholds or other appropriate inputs. Similar patterns are provided for the other objects of interest (in our case, food source, entertainment, sleep area and concerns). Figure 7.23 shows example uses of these patterns in an action-selection hierarchy.

It should be noted that we do not simulate true visual sensing or object recognition but instead store the information about an object with each object in the environment. However, the sensing of objects is related to the current location and orientation of the character’s head so in that sense, where a character is looking does determine which objects the character senses.

7.6 Motion Model

The lowest-level of motion control for virtual characters is not so much a factor of the character’s decision process as it is the physical laws of the environment the character inhabits. Therefore, the specification of this level of control is usually

integrated with the structure of the character, either in the form of physics-based models or other geometric constraints. Physics-based modelling provides some of the most sophisticated facilities to date for controlling low-level motion. Some of the drawbacks of physics-based models are that they are complex and are highly optimized for a particular geometry. Hence, creating a physics-based model of a character is difficult and time consuming.

One of our goals is to illustrate that the ViB model can be used to provide the low-level motion model for a character. We are not concerned with providing the same type of motion control as a physics based model since this would be no less effort than using physics-based modelling and probably would not perform as well due to the reactive nature of the ViB model. Instead, we aimed to provide a low-level motion model that can be used to provide adequate motion that is not specific to a particular geometry and is suitable to be used for testing higher-level behavior layers. At the same time, we aimed to provide a low-level motion model that is loosely coupled to the higher-level behavior layers in order to allow more sophisticated models to be substituted with as little modification as possible.

In providing these facilities, we have assumed a minimal amount of support from the underlying architecture supporting the structural model. Therefore, we have built motion models that rely on the availability of end-effector or contact point positions and points of intersection between lines of motion and surfaces in the environment. Although the availability of more sophisticated information would improve the motion models, this information is enough to provide believable, if not realistic, looking motion in many situations.

In order to determine the location of a body, whether it is a character or some other object, we use kinematic techniques that involve acceleration and inertia. We also incorporate the output of constraint controllers in the location calculation to handle collisions and resting contact.

7.6.1 Kinematic Motion

The location of character within its environment is determined using the kinematic equation of motion which consists of an inertia and an acceleration component. The location of a body at time t is

$$location_t \stackrel{\pm}{=} \Delta location_{t-1} + acceleration_t \Delta t^2 / 2$$

This location determination equation works well when there is no surface contact or collisions involved. However, when these events occur, the sudden change in acceleration causes the above equation to become invalid. Therefore, we expand on this model of motion in order to allow impact to be incorporated into the location calculation. The kinematic motion model we use is implemented as a pattern library that is an extension of the `KinematicDisplacementChannel` presented in Section 5.1.4. In this model, the determination of the character's *location* channel is divided into several sub-channels.

$$\begin{aligned}
 location_t &\stackrel{\pm}{=} location.delta_t + \sum_i adjustment_i \\
 location.delta_t &= \Delta location_{t-1} + location.acceleration \Delta t^2 / 2 \\
 location.acceleration_t &= (0 \ -9.8 \ 0) \\
 location.velocity_t &= location.delta_t / \Delta t
 \end{aligned}$$

Contributions representing standard acceleration are made to the *acceleration* channel and in our case this amounts to the acceleration due to gravity. Effects that cause abrupt changes in location, such as a collision, are not made to the *acceleration* channel because the sudden acceleration due to these effects causes the location calculation to be unstable. Instead, they are made directly to the *location* channel itself as *adjustments* representing a change in location due to the impact.

The *velocity* channel is provided for feedback purposes for use by higher-level behaviors. Because of this, we do not use the actual change in location, $\Delta location_t$, to calculate the velocity but instead use the change due to any accelerations acting

on the body. This allows higher-level behaviors to react to the perceived velocity during an impact even as the location is being adjusted to compensate for impact. Therefore, a *delta* channel is used to calculate the effects of acceleration and inertia on the location and it is this channel that is used to calculate the velocity.

7.6.2 Surface Contact

The goal of the SurfaceContact controller is to ensure that all parts of a character remain on or above the surface in question. In a reactive model such as ours, it is not possible to ensure this at all times but when interpenetration does occur, we can adjust the character's location such that any interpenetration is removed as quickly as possible while still providing realistic looking animation.

Surface or resting contact is handled by contributing an adjustment to the location channel of the character. In order to determine and compensate for surface contact and interpenetration, we make use of a set of contact points that have been defined for the character. For each contact point, the location and amount of penetration into a surface is provided by geometric sensors. The maximum amount of penetration for all contact points is used to compensate for any penetration. Therefore, the adjustment due to penetration is simply the maximum amount of penetration negated.

$$adjustment_{penetration} = -d_p$$

where d_p is a vector representing the distance and direction of maximum penetration. The application of this adjustment to the location channel will move the character back to a position where no interpenetration occurs.

Because location determination involves the use of dynamic characteristics, namely the location delta, the above adjustment will affect how the location is determined on future ticks. In other words, if the above penetration adjustment is applied directly to the location determination of the character, the result is equivalent to a rebound or bounce with no energy loss. In order to compensate for this in a visually acceptable way, any change in location of the character while the character is in contact with the surface is damped. The direction of the force that is pulling the character to

the surface is used as the direction of damping. Consequently, the surface contact adjustment due to damping is

$$adjustment_{damping} = (-damping \cdot (\Delta location_{t-1} \cdot v_f)) \cdot v_f$$

where *damping* is the scale factor between 0 and 1 used to damp the change in location when the body is in contact with a surface, v_f is the direction of the downward force that is causing the character to contact the surface, and $\Delta location_{t-1} \cdot v_f$ is the change in location for $t-1$ in the direction of v_f . We use a damping factor of 0.6 and determine v_f using the acceleration such that $v_f = location.acceleration / |location.acceleration|$

A similar approach to the above surface contact determination can be used to determine collisions in the direction of motion instead of the direction of force by using the velocity to determine v_f instead of the acceleration. A similar approach to resting contact and forward collision detection is used in [4].

The advantages to this approach to providing collision and resting contact is its low computation cost and generality. However, there are several drawbacks. First of all, because it is reactive, there is inevitably some interpenetration at impact. The amount of interpenetration depends on the velocity at impact and the frame rate. Also, the method relies on the use of a finite number of contact points. Therefore, contact with areas of a character where there are no contact points will go unnoticed. Finally, there is no reaction to the impact built into the character as is the case with physics based models. Therefore, higher-level behaviors will need to be added to the character to react properly to a collision.

The bottom line is that this approach is no substitute for physics-based modelling. However, in the absence of such a model, or other models that provide similar capabilities, this approach does provide results that are visually acceptable in most situations. The quality of motion produced can also be improved by the inclusion of higher-level behaviors that anticipate contact and hence react ahead of time as shown in Figure 7.9 or that avoid collisions when possible.

Figure 7.9: Absorbing Collision Impact

7.6.3 Surface Friction

The second aspect of surface contact is the motion that results due to friction between contact points and the surface. For each contact point that is in contact with a surface, any lateral motion of the point is used to generate motion for the character. Therefore, adjustment due to friction on the location of the character is

$$adjustment = \sum_c -\omega_{ct}(\Delta location_{ct} - v_f(\Delta location_{ct} \cdot v_f))$$

where $\Delta location_{ct}$ is the change in location of contact point c and v_f is the direction of the downward force maintaining the contact. This results in a vector that is the inverse of the contact point motion perpendicular to the downward force.

The weight, ω_{ct} , for contact point c is

$$\omega_{ct} = \begin{cases} 1 & \text{if } c \text{ is in contact at both } t \text{ and } t-1 \\ 0 & \text{otherwise} \end{cases}$$

However, multiple contact points moving in the same direction will cause extra motion and contact points moving in different directions may cause slight oscillations. Therefore, the friction contribution for each contact point are combined using

normalized weighting from Section 6.1.2 which results in an averaging of the effects of multiple active contact points. We have not included a coefficient of friction in this calculation since there is often no slipping between contact points and a surface during motion.

Figure 7.10: Friction Causing Motion

One of the advantages of approximating surface friction is that higher-level behaviors such as actions need only specify the joint angles for a character and any corresponding motion will be generated through contact point contact and friction. Again, because it is a reactive model, there is an initial slip when the character starts moving but the results are usually acceptable unless the motion starts abruptly. One restriction is that actions that generate motion must provide the proper joint angles over time to achieve the proper motion. However, this is also the case for actions that are controlling a physics based model as well. Figure 7.10 shows a walking motion that relies on friction to move the character forward.

7.6.4 Other Constraints

The surface contact and friction controllers handle a wide range of situations. However, because they are only approximations, inevitably, specific situations arise for which they do not provide the necessary low-level motion control. Suitable constraint

controllers can be hand crafted to deal with these particular situations. For instance, we have developed additional constraint controllers for sitting and other types of specialized interactions. Coordination of constraint controllers and higher-level behaviors is handled by the posture controller which is presented in the next section.

7.7 Controller Model

The controller model acts to connect the motion, action and behavioral models together. It does this in two ways. First, action controllers are used to map geometric state information and higher-level behavior input into output that is used by actions to determine their effects on the DOFs of the geometry. Second, it acts to communicate restrictions on motion through the use of a posture controller.

7.7.1 Action Control

The ActionController library of AniMos provides the capability of defining custom action controllers. ActionControllers receive input from higher-level behaviors and map these inputs into outputs that control one or more actions. Generally, action controllers are hand-crafted to serve a particular purpose. However, we have identified two general types of controllers, namely targeting controllers and coordinating controllers and have created custom pattern libraries to support their creation.

Targeting Controllers

Targeting controllers are used to convert a target location into one or more angles that represent the rotation about a particular axis required to point a reference direction towards the target location. The effects of targeting controllers are divided into one or more targeting dimensions which are used to calculate the desired angles. These angles can then be used by actions to determine the effects on the DOFs of the character.

A targeting dimension is a plane defined by an origin p_r and a normal n_r and also consists of a reference direction v_r that is parallel to the dimension's plane. The

Figure 7.11: Targeting Controller Dimension

output of the dimension is the angle θ that the direction v_r must be rotated about axis n_r in order to be aligned with the projection of a target point, p , on to the dimension's plane, as shown graphically in Figure 7.11. Thus, θ is determined as follows.

$$\begin{aligned} v &= (p - p_r) - n_r((p - p_r) \cdot n_r) \\ \theta &= \mu \arccos(v \cdot v_r) \\ \mu &= \begin{cases} 1 & \text{if } v \times v_r = n_r \\ -1 & \text{otherwise} \end{cases} \end{aligned}$$

where μ uses the cross product to determine the direction of rotation required about n_r .

Figure 7.12 shows an outline of the TargetingController library. When this library is used, the “inputs”, “outputs” and “effects” variables defined in the library will be used as the input variables to ActionController (whose definition and use is described in Section 6.2.2). Defining these variables as patterns in the library allows child libraries to override them as needed, as we will see below. The Dimension pattern uses the TargetingDimension pattern and the variables “targetLocation”, “dimensions”, “normals”, “referenceLocations” and “referenceDirections” to create the appropriate subsystem for each of the specified dimensions in order to perform the targeting calculation described above. The details of these patterns are not shown but appropriate input values for the mentioned variables are shown in the HeadTargetingController pattern in Figure 7.13.

```

Library TargetingController {
  TargetingController = Pattern ActionController{}
  effects = Pattern [Dimensions{}]
  inputs = Pattern [
    MemoryVariable{
      variableName = "targetLocation"
    }
    EnablingVariable{
      enablingScheme = Pattern EnabledWhileActivated{}}]
  outputs = Pattern []
  Dimensions = Pattern ...
  TargetingDimension = Pattern ...}

```

Figure 7.12: TargetingController Definition

The TargetingController library has been used to provide several different controllers including a BodyTargetingController, HeadTargetingController, LegTargetingController and HandTargetingController. The definition of HeadTargetingController shown in Figure 7.13 uses two dimensions to provide the angles required to orient the head to look at a particular location. The outputs of the HeadTargetingController are used by the LookAt action to look at a particular location as shown in Figure 7.14. Because TargetingController is a library, it can be overridden to provide additional inputs, outputs or effects as required. The HeadTargetingController provides the outputs leftTurnAngle and upTurnAngle that allow access to the angles of the two targeting dimensions.

Coordinating Controllers

Coordinating controllers provide output that is used to coordinate the effects within an action or the effects across multiple actions. The need for coordination is mostly related to cyclic motions. Therefore, we have created a general CycleController library that converts a desired input tempo into a cyclic phase output that can be used as a coordination signal.

$$phase_t = (phase_{t-1} + tempo_t(2\pi\Delta t)) \mod 2\pi$$

```

Library HeadTargetingController extends TargetingController {
  HeadTargetingController = Pattern TargetingController{
    controllerName = "HeadTargetingController"
    dimensions = ["horizontal" "vertical"]
    normals = [upDirection{} leftDirection{}]
    referenceLocations = [headLocation{}]
    referenceDirections = [forwardDirection{}]}
  effects = Pattern super.effects{} + {
    SetChannelToReplaceOutput{channel = ["leftTurnAngle"]}
    SetLeftTurnAngle{value = channelOutput(["horizontal" "angle"], 0)}
    SetChannelToReplaceOutput{channel = ["upTurnAngle"]}
    SetUpTurnAngle{value = channelOutput(["vertical" "angle"], 0)}
  }
  outputs = Pattern super.outputs{} + {
    ChannelVariable{variableName = "leftTurnAngle"}
    ChannelVariable{variableName = "upTurnAngle"}}}

```

Figure 7.13: HeadTargetingController Definition

Like the TargetingController, the CycleController is a library that can be extended in order to provide specific controllers. Example controllers include the BodyCycleController and the ScanningController. The BodyCycleController is used by actions such as Walk to coordinate the motion of the legs and arms. Other actions that are coordinated with walk such as running or stepping to a particular location also use the phase output of this controller. The ScanningController coordinates head motions while a character investigates an area in order to scan the complete area in a deliberate fashion.

7.7.2 Posture Control

There are several uses for a posture controller, as stated in Section 6.2.2. First, the posture controller acts to communicate the current posture of the character to higher-level behaviors in order that they may determine the proper actions to perform. Also, the posture controller controls when a change in posture takes place. Finally, each posture has a set of effects that are active while the character is in that posture.

For Robby, we use the posture controller to coordinate the low-level motion model

Figure 7.14: Looking at a Chair

provided by the constraint controllers presented in Section 7.6 with the ActionModel presented in the next section. The effects of each posture are used to enable the required constraint controllers for that pose along with any default actions that will place the body in the proper pose. In essence, the posture controller acts as part of the low-level motion model by ensuring that the DOFs of the character are set to appropriate values in the absence of other actions affecting those DOFs. Because of this, there is no need for an action to return DOFs to a neutral value, as done in [11] and the higher-level action organization need not be concerned with specifying values

Figure 7.15: Posture Model for Robby

for all DOFs, as is required for the lowest layers in the action group hierarchy of [54]. A posture can also activate actions that restrict the effects of other actions in order to control what actions can be performed when in a particular posture. The action model organization that supports this is presented in the next section.

```

PostureModel = Pattern PostureController{
  startingPosture = "Standing"
  postures = Pattern [
    ...
    Posture{
      postureName = "Moving"
      effects = Pattern [
        EnableController("SurfaceContactController")
        EnableController("SurfaceFrictionController")
        ActivateAction(["PostureActions" "Stand"])]
      transitions = Pattern [
        Transition(not(wantMoving{}), "Standing")
        Transition(wantFalling{}, "Falling")]
    ...}]

```

Figure 7.16: The Moving Posture

For Robby, the PostureController pattern is used to define the posture model shown graphically in Figure 7.15. Transitions between postures are triggered by conditions based on character state information and requests from higher-level behaviors. Figure 7.16 shows the specification of the Moving posture. The transition from Standing to Moving occurs when a higher-level behavior requests the transition and the Moving posture is maintained as long as this higher-level behavior continues to request the posture or the Falling posture is requested. The transition from Moving to falling could be triggered by the downward velocity of the character. Instead it is triggered by a higher-level behavior that detects that the character is falling and requests the posture change. The Falling behavior, which is mentioned in Section 7.9.4, anticipates the landing in order to activate the landing motion before contact is made to improve the action timing.

Some postures, such as Sitting as shown in Figure 7.17, are maintained until

Figure 7.17: Robby Having a Rest

another posture is requested. These different transition characteristics are meant to differentiate stable body positions versus those that require effort to maintain. A character will remain sitting until it desires to get up but a moving character will stop once the desire to move is lost. Other postures are transitional postures that are used to free higher-level behaviors from the necessity of knowing how to transition between postures and to indicate to higher-level behaviors that a transition is taking place.

7.8 Action Model

As stated in Section 6.2.3, the purpose of the action model is to map controller output to effects on the parameters or degrees of freedom that control the underlying structural model. For our characters, these parameters represent joint angles and, therefore, actions map the output of the controllers presented in the previous section to the effects on these joint angles. Figure 7.18 shows the action model we implemented for Robby using the ActionModel library presented in Section 6.2.3. We used the Action pattern to implement the individual actions along with ActionLayer to define relationships between these actions.

Figure 7.18: Action Model of Robby

7.8.1 Action

Actions are hand-coded to translate controller output to the desired effects on the geometry. Implicit in the definition of an Action is the activation level that is triggered by a higher-level entity using the `ActivateAction` pattern and is used by the `SetDOF` and `SetDOFDelta` patterns to weigh the provided DOF values appropriately. We have implemented several actions for Robby for moving through the environment (e.g. “Walk”, “Turn”) and interacting with the objects in them (e.g. “LookAtObject”, “Sit”, “ReachWithRight”).

Figure 7.19 shows a simplified walk action. Most of the configurations in the “effects” pattern set certain DOFs of the character’s legs and arms in order to generate a walking motion. The “phase” from the “BodyCycleController” is used to coordinate the motion. For the left arm and right leg, the phase is offset by π in order to get the proper leg and arm coordination for a walking motion¹. Without going into the details, the effect on the [“orientation” “yaw”] channel represents a desired turn angle. The turn angle is determined using the output of the “BodyTargetingController”. The inclusion of turning behavior within an action is discussed below.

¹The joint angles for our character are controlled using scalars in the range -1 to 1.


```

Walk = Pattern Action{
  actionName = "Walk"
  phase = Pattern GetControllerVariable("BodyCycleController", "phase")
  effects = Pattern [
    SetDOF(["Right" "Shoulder" "pivot"], sin(phase{ }) * 0.2)
    SetDOF(["Right" "Elbow" "flexion"], sin(phase{ }) * 0.4)
    SetDOF(["Left" "Shoulder" "pivot"], sin(phase{ } + pi) * 0.2)
    SetDOF(["Left" "Elbow" "flexion"], sin(phase{ } + pi) * 0.4)
    SetDOF(["Right" "Knee" "flexion"], cos(phase{ } + pi) * 0.5)
    SetDOF(["Right" "Hip" "pivot"], sin(phase{ } + pi) * 0.2)
    SetDOF(["Left" "Knee" "flexion"], cos(phase{ }) * 0.5)
    SetDOF(["Left" "Hip" "pivot"], sin(phase{ }) * 0.2)
    SetDOFDelta(["orientation" "yaw"],
      GetControllerVariable("BodyTargetingController", "leftTurnAngle"))]]}

```

Figure 7.19: A Simplified Walk Action

7.8.2 Action Layering

Using the `ActionLayer` pattern, Actions are arranged in layers based on the scope of their affect on the character. An example `ActionLayer` specification is shown in Figure 7.20. Each layer is set up such that only a single action is active at a time. It is the order of the actions in a layer that determines their relative priority. For instance, if both the Walk and Turn actions are being activated, then the Walk action will subsume the Turn action. This is not to say that a character cannot walk and turn at the same time. The specifics of turning are related to each specific locomotion action. Therefore, each of these actions includes effects that will turn the character. The desire to turn is communicated to these actions by the `BodyTargetingController` and each action that incorporates turning uses the output of this controller to determine the amount of turn required. However, it is not appropriate to turn during some actions, such as absorbing the impact of landing after a fall. Therefore, some locomotion actions do not incorporate any turning effects.

This type of action grouping simplifies the higher-level behaviors that control the actions. For instance, a behavior to face a particular location need only set the target location variable of the `BodyTargetingController` and activate the Turn action.

```

LocomotionLayer = Pattern [
  ActionLayer{
    layerName = "Locomotion"
    actions = [AbsorbFall Stopped StepToLocation Walk Turn]
    LayerActivationControl = Pattern OneActive{}}
  ContributeSourceLayerToTargetLayer{
    sourceLayer = "Locomotion"
    targetLayer = []}]

```

Figure 7.20: The Locomotion Action Layer

If no other locomotion actions are active, the character will turn on the spot. If a higher-priority locomotion action is active, the Turn activation will be ignored and the character will turn in a fashion dictated by the higher-priority action. In either case, the “face location” behavior is not concerned with the mode of locomotion, only that at least one is active and that the character turns to face the desired location. In those cases when the active higher priority locomotion action does not support turning, it is up to either the “face location” behavior or other higher-level behaviors to identify and deal with the situation. The PostureController presented in the previous section helps to identify such situations.

Our action grouping and layering is similar to that used by Perlin in the Improv system [54, 55]. However, unlike Perlin, we do not enforce a subsumption ordering between the layers. For the locomotion and looking layers, the DOF channels of each layer are contributed to the corresponding DOF channel in the main DOF layer using the activation of the layer channel along with normalized weighting as presented in Section 6.1.2, resulting in a weighted blend of contributions to overlapping DOFs. One layer can be given more influence than another by using strength adjusted weighting instead. For instance, the looking layer is assigned a higher strength than the locomotion layer to allow looking actions to have more affect on any overlapping DOFs. However, each of these DOFs may still be partially influenced by an active locomotion action.

7.8.3 Posture Actions

We do make use of a subsumption ordering between the posture actions layer and the effects of the other actions. The effects of the posture layer are not blended with the effects of the other layers but instead provide the default values for each DOF in the DOF Layer (using the default contribution pattern associated with a contribution weighting scheme as described in Section 6.1.2). Therefore, only if no other actions or layers are affecting a DOF will the values provided by the posture layer be used. The appropriate action in the posture layer is activated when the corresponding posture is active in the PostureController. Since the active posture action only has an effect when there are no other active contributions to the DOF, the action acts to provide a suitable default pose. In essence, these actions act to compensate for the lack of geometric constraints that would place the body in an appropriate pose when no actions are influencing it. However, the posture controller and corresponding actions also provide actions that transition between postures which simplifies higher-level behaviors that trigger these transitions.

In addition to the default action, a Posture of the PostureController can activate actions in other layers to ensure that invalid actions do not take place in a particular posture. For instance, the Stopped action has a high priority within the locomotion layer and is activated when the current posture restricts motion. For instance, when the current posture is Sitting, the character should not be able to walk. If a higher-level behavior wants to trigger a walk, it must first trigger a posture change through the PostureController. It can be argued that a character should be able to walk in a restricted fashion while sitting and, more generally, that motion combination should not be overly restricted at the action level. However, less restrictions at the action level leads to more complexity at the action-selection level and, with less restrictive action models, errors in the action-selection process can lead to undesirable character motions. Therefore, we make use of these restrictions to improve the robustness of the overall animation model of the character.

7.9 Behavioral Model

The behavioral model of a character is concerned with action-selection which involves translating the character's internal state and environmental conditions into a particular set of actions in the action model that should be activated. In order to provide this, we have divided the behavioral model into two components as shown in Figure 7.21. This is similar to the organization used by Tu [71] to provide behaviors for artificial fish. These models make use of the desires of the creature and sensory input to determine an appropriate course or sequence of action.

Figure 7.21: Behavioral Model Layering

7.9.1 Desires

An important aspect in performing action-selection is the desires of the character. We represent the desires of a character using a set of behavior attribute variables that are similar to the internal variables used by Blumberg [11]. Each variable has a *growth* and *damping* factor that is used to determine how the value of the attribute changes over time.

$$attribute_{it} \stackrel{\pm}{=} -damping(attribute_{i(t-1)}) + growth + \sum_j \alpha_j gain_j$$

In addition, certain behaviors, when active, affect the value through a *gain* that is weighed by the behaviors activation level α .

As can be seen in Figure 7.6, Robby has attributes representing hunger, fatigue, need for sleep, boredom and concern. The BehaviorAttribute pattern library is an extension of ArticulatedVariable that overrides the InitializeVariable pattern to apply

the growth and damping and the `SetVariable` pattern to apply a gain to the attribute. The `BehavioralModel` pattern provided by `AniMoS` uses a `VariableSystem` to allow the sub-model to access and modify behavior attributes using pattern names customized to each attribute.

7.9.2 Sensory Information

Another important factor in action-selection is sensory information about the surrounding environment and the objects in it. The level of intelligence that a virtual character appears to have is directly proportional to the character's ability to sense information about its environment. From a ViB model standpoint, we are not concerned with how the sensory information is generated but, instead, we are concerned with how it is incorporated into a model. Therefore, sensors are specified as part of the character's geometry and the output of these sensors is provided to the character's ViB model.

There are two levels of sensing provided by the Geometry component. The first level, which corresponds to the state feedback model presented in Section 7.4, provides low-level geometric state information such as distances to obstacles and locations of end-effectors. The second level of sensing, described in Section 7.5, provides information about objects in the environment that is used by the behavioral model to choose appropriate courses of action. For Robby, we make use of common items such as a chair, bed, fridge, and TV. The sensing of these items, combined with the desires presented in the previous section will be used to identify an appropriate course of action, be it to rest, sleep, eat or watch TV.

7.9.3 Intention Generation

The main pursuit of a character is determined using a behavior hierarchy similar to that proposed by Blumberg [11]. However, this behavior hierarchy is not used to provide the complete action-selection mechanism of the character but is instead used to determine the character's current intention. This is similar to the intention generator used by Tu [71]. Tu's intention generator determined the main course of

action for the artificial fish and provided the necessary information to the lower-level behaviors. In the same way, we use a behavior hierarchy to select an appropriate course of action based on the character's internal state or desires and the sensed objects in the environment and to provide the necessary sensor information to the behavior modules that will fulfill the triggering desire.

Figure 7.22: Intentions Model of Robby

Figure 7.22 shows an outline of the intentions model for Robby. The top level of this model is concerned with deciding which of the basic desires of the character is to be pursued. Each desire has an associated set of dependencies which are used to determine the value of that desire. In Figure 7.23, the value of the *Fatigued* behavior is determined by the sum of the three dependencies of the Behavior as described in Section 6.2.4. The presence and nearness of a rest area acts to increase the desire to rest. Generally, the behaviors associated with the other desires have similar dependencies. These behaviors are grouped together in a *BehaviorGroup* in order that only one of them will be active at a particular time.

When a particular behavior is active, its children are also activated. The children of each desire determine the specifics of how to fulfill the desire. Although the fulfilling of each desire has their own peculiarities, they are generally similar to that of *Fatigued* as shown in Figures 7.22 and 7.23. Each of the behaviors in the lower-level make use of sensor information and weighting to determine their value and depending on this information, a particular course of action will be selected. The course of action may be to modify behavior attributes or to enable modules in the lower-level action-sequencing layer. Along with enabling modules, the behaviors also provide the

information required by these modules to perform their tasks.

```

Behavior{
  behaviorName = "Fatigued"
  dependencies = [
    fatigued{}
    fatigued{} * restAreaPresent{}
    fatigued{} * nearRestArea{}
  ]
  effects = Pattern [BehaviorGroup{children = Pattern [
    Behavior{
      behaviorName = "EnterRestArea"
      dependencies = [atRestArea{} * 10]
      effects = Pattern [
        EnableModule("EnterRestArea")
        SetModuleInput("EnterRestArea", "restArea", sensedRestArea{})
      ]
    }
    Behavior{
      behaviorName = "ApproachRestArea"
      dependencies = [restAreaPresent{}]
      effects = Pattern [
        EnableModule("ApproachObject")
        SetModuleInput("ApproachObject", "object", sensedRestArea{})
        SetModuleInput("ApproachObject", "type", restAreaIdentifier{})
      ]
    }
    Behavior{
      behaviorName = "FindRestArea"
      dependencies = [0.1]
      effects = Pattern [
        EnableModule("FindObject")
        SetModuleInput("FindObject", "type", restAreaIdentifier{})
      ]
    }
  ]
  ]
}

```

Figure 7.23: Fatigued Action-Selection Group

It should be noted that the FindObject and ApproachObject modules referenced in Figure 7.23 are general in the sense that they can look for or approach most objects. However, for some objects, the standard means of performing these tasks may not be appropriate. Therefore, these modules are designed in such a way that their effects can be subsumed by modules that know the specifics of how to look for or approach a particular object. The same is true for the EnterRestArea module which will generally try to sit on objects but should be overridden by another module for rest areas that require different handling. This process is described in the next section.

7.9.4 Action Sequencing

For our character, we perform action sequencing using a network of interconnected modules similar to those proposed by Brooks [17]. Like Brooks, we make use of FSMs within the modules to determine how the module input is translated to module output. However, in some cases, we make use of behavior hierarchies like those used in the previous section instead. The inner workings of these modules are not as important as how they are connected to form the behavior network. The BehavioralModel and other AniMoS libraries provide the patterns used to create the action sequencing network.

The action sequencing behavior networks of Robby consists of sub-networks for finding, approaching and interacting with objects. The intention generator only deals with general object types such as food sources, rest areas, entertainment sources and so on. In some cases, a generic behavior sub-network is adequate for dealing with a general object type. However, in many cases, a specific means of dealing with an object is required. Instead of adding these special cases to the intention hierarchy, we use subsumption layering as proposed by Brooks to override the generic sub-networks when required. In the following sections, we will give an example of a generic network for approaching an object and also describe how we support the overriding of this network for specific object types.

Behavior Network

Figure 7.24 shows the sub-network used to approach an object. When a particular branch in the intention generator is active, such as “ApproachRestArea” in Figure 7.23, the ApproachObject module is enabled and provided with the object to be approached and an object type that identifies the reason why the object is being approached. The object provided to the module is a reference to a ViB model that contains the information about the sensed object that can be used to perform the necessary actions.

Figure 7.25 contains the specification of the ApproachObject module. The module simply passes on the provided object and type and uses a generic means to determine

Figure 7.24: Network to Approach an Object

the location to be approached from the provided object information. The targetLocation output is connected to the ApproachLocation module along with the approaching output which is used to enable ApproachLocation. ApproachLocation combines turning and moving as required to get close to the target location and then closes in to the location when the character is close enough. The low-level modules used by ApproachLocation set appropriate controller variables and activate actions in the action model to move and turn the character. These low-level modules also use the posture controller to query the current posture to ensure that the character is in the proper posture to carry out the desired action. Figure 7.26 shows Robby approaching a chair using the ApproachObject sub-network.

Most module connections are similar to those used by Brooks and communicate information in the form of a message that is sent at a particular time and buffered in the receiving module. However, in order to improve efficiency during ViB model evaluation, we use connections to enable modules as well as communicate information. Therefore some outputs are activation signals that send a continuous signal that is used to enable another module for the period when the signal is active. Most of the modules in the action sequencing layer are enabled either directly or indirectly in response to the current intention.

```

ApproachObject = Pattern Module{
  moduleName = "ApproachObject"
  EnablingScheme = Pattern EnabledWhileActivated{}
  inputs = ["object" "type"]
  outputs = ["target" "targetType" "approaching" "targetLocation"]
  approaching = ActivationVariable
  effects = Pattern [
    FiniteStateMachine{
      initialState = "Start"
      states = Pattern [
        State{
          stateName = "Start"
          effects = Pattern []
          transition = Pattern EventDispatchTransition{
            events = Pattern [EventTransition(not(object{}==zero), "Focus")]
          }
        }
        State{
          stateName = "Focus"
          effects = Pattern [
            SetTarget{value = object{}}
            SetTargetType{value = type{}}
            SetTargetLocation{value = sensedObjectEntryPoint(object{})}
            SetApproaching{}}
          transition = Pattern ImmediateTransition("Approach")
        }
        State{
          stateName = "Approach"
          effects = Pattern SetApproaching{}
          transition = Pattern EventDispatchTransition{
            events = Pattern [EventTransition(not(object{}==target{}), "Start")]
          }
        }
      ]
    }
  ]
}

```

Figure 7.25: ApproachObject Module

The Module library provided by AniMoS and described in Section 6.2.4 does support the ability to specify that a module should be enabled until it is explicitly disabled or that it should always be enabled. The former ability can be used to allow a Module to continue affecting the behavior of the character for a period after it is no longer receiving an enabling signal. The later ability allows modules to react to certain conditions at any time regardless of the current intentions of the character. An example of this is the Falling module which is always active in order to detect when a character is falling and also trigger the appropriate landing action when the

Figure 7.26: Approaching A Chair

character lands.

Subsumption

The general means of overriding the ApproachObject module is shown in Figure 7.27. The target and targetType outputs make the object and its type available for inspection by other modules. An overriding module uses this information to determine if it should override the default approach behavior. If it is determined that the default approach behavior should be overridden, the overriding module inhibits the target location provided by the ApproachObject module and passes the proper desired location to the ApproachLocation module.

In order to aid in the creation of modules that override ApproachObject, we have implemented an ApproachObjectOverride pattern library that provides the relationship shown in Figure 7.27. This library provides the necessary inputs and outputs and defines the connections to the other modules involved. There are several ways this library can be used to create an overriding module. If only a different location determination is required for an object, the library can be configured using several variables that indicate the specifics about the object to which the modules applies and the function used to determine the approach location. If more specific approaching

Figure 7.27: Overriding Approach Object

behavior is required, the library can be overridden to tailor the behavior as needed.

```

ApproachTV = Pattern ApproachObjectOverride{
  moduleName = "ApproachTV"
  desiredObjectType = entertainmentIdentifier{}
  desiredObjectID = tvIdentifier{}
  desiredTargetLocation = Pattern externalChannelOutput(object{}, ["OnSwitch" "location"])
    + (externalChannelOutput(object{}, ["direction" "forward"]) * 0.3)
  desiredEffects = Pattern FiniteStateMachine{
    initialState = "Test"
    states = Pattern [
      TestState{}
      ApproachState{
        toState = "TurnOnTV"
      }
      State{
        stateName = "TurnOnTV"
        ... details dealing with this state ...
      }
    ]
  }
}

```

Figure 7.28: ApproachTV Module

Figure 7.28 shows how we used this library to create the ApproachTV module. A finite state machine specific to TVs is provided but the TestState and ApproachState provided by the extended ApproachObjectOverride library are used (see the pattern that variable "states" is bound to). These states make use of the provided object identification information and target location to override the ApproachObject module

when the object being approached is a TV. Once the character is at the TV, the custom portion of the effects is used to turn on the TV, as shown in Figure 7.29

Figure 7.29: Approaching the TV

7.10 Results

In this section, we summarize the various sub-models used to construct of virtual character and present our results.

7.10.1 Appearance

Our main goals are not directly related to a character's appearance. Consequently, we have created simple geometric models for the characters and objects that inhabit our test environment. From our standpoint, what is important about a geometric model is the parameters used by higher-level models to control it. The patterns representing the geometric model of a character or object are used to both generate the appearance and provide the required information and control to higher-level models.

7.10.2 Motion

In our architecture, low-level motion is provided by a kinematic model consisting of motion constraint controllers that incorporate physical notions. Examples include the surface contact and surface friction controllers presented in Section 7.6. Although the animation produced does not match that of true physics-based models for realism, these controllers are less complex and are not specific to a particular geometric model. However, each controller may only work properly when used in conjunctions with appropriate actions and in the intended situations. Consequently, the motion produced using these controllers is acceptable in the situations for which they were intended but unanticipated interactions can cause unrealistic or undesirable effects. The posture controller is key in identifying when particular motion controllers should be used. The robustness of motion controllers can be greatly improved by controlling the type of environment they are used in and by providing better low-level collision detection.

Motor skills are provided by the actions and action controllers presented in Section 7.7 and 7.8 respectively. The separation between action controllers and the actions themselves serves two purposes. For one, it allows multiple actions to be coordinated through the use of shared controller output. Secondly, it separates the value determination of the high-level control parameters for a motion from that of the low-level parameters that control the geometry. Thus, action controllers act as an abstraction barrier between higher-level behaviors and actions while actions act to map controller output to the appropriate affects on the geometry.

One of the requirements of an action is that it work properly in conjunction with the corresponding motion controllers. For instance, a walking action must raise and lower the feet properly in order to cause the surface friction controller to generate forward motion. The same requirement exists with motion that is provided by a physics-based model. Certain actions, such as walking to a particular spot, require extremely sophisticated control that can be difficult to provide. An advantage of our approach is that special controllers can be created to compensate for insufficiencies in any actions.

Actions can be arranged in groups and layers. Action grouping helps reduce

the complexity of higher-level behaviors by enforcing certain activation restrictions between the actions in a group. Action layering helps determine how the effects of multiply active actions can be combined or blended properly.

7.10.3 Behavior

Our behavioral model maps sensor input to action controller input and action activations. It is made up of two components, the intention generator, which determines an overall course of action for the character, and the action sequencer which determines the required controller input and action activations to fulfill the chosen course of action. These components are built using well known behavioral model constructs such as action-selection trees, finite state machines and subsumption networks. However, what is unique about our model is the integration of these constructs within a single behavioral model.

The ViB model representation is particularly well suited to behavioral models as they are typically reactive in the sense that the course of action chosen is a function of the current sensor input and the internal state of the character. In addition to the state of behavior components such as actions and modules, the internal state of a character consists of a set of desire levels modelled as scalar variables. However, there are no planning capabilities that make use of knowledge or previous experience. These facilities would be provided by a higher-level cognitive model. Sensor input is used to provide both information about the environment and information that would be provided by such a higher-level model.

7.10.4 Response Time

The response time of our architecture consists of two aspects; the time to compile a model from a specification and the frame rate of the resulting ViB model when used to animate a virtual character. The time required to compile a specification is directly proportional to the size of the model. The time required to compile the model presented in this chapter is 2 to 4 minutes. This is acceptable when models are compiled infrequently which is the case for models that are being used for a particular

application. However, the length of a compile can be detrimental during the creation of models, especially since behavioral models need a fair bit of tweaking to arrive at the desired results. We have improved the situation somewhat by supporting the incremental compilation of high-level sub-models. For example, once the entire model is compiled once, the various sub-models, such as the action model or behavioral model, can be compiled independently (although certain changes still require a full recompile). In the future, we would like to extend incremental compilation to lower-level components such as individual actions or modules.

The execution time of a ViB model is proportional to the size of the model. As model size increases, so does the amount of time required to evaluate the model on each tick. The scalability of ViB models has been improved by the ability to disable dormant systems. However, this facility is not automatic but instead relies on proper model structuring provided by the specification patterns. For our implementation of the supporting architecture, the model we presented in this chapter, which consists of 1440 channels and 660 systems, runs at 4 to 6 frames/second on 450 MHz Pentium and 10-12 frames/second on a 1000MHz Pentium. Due to the reactive nature of our execution model, we have found that a minimum frame rate of 20 frames/second is required to produce acceptable results (This requirement has been observed by others as well [11]). Therefore, our example runs at between quarter and half time which still allows us to visually evaluate the resulting behavior.

7.10.5 Design

Specifying models using pattern oriented composition involves a different way of thinking than when using a programming language. The focus is on hierarchical composition instead of algorithm creation. The available pattern primitives are essential to the success of the approach and play a large role in determining the resulting model. We have been able to build ViB models at higher-levels of abstraction using a relatively small number of primitive patterns. Furthermore, the layering of the higher-level abstractions developed support the incremental creation of behavior models.

Dynamic variable scoping and the ability to provide smaller-scale patterns as input to a pattern add flexibility to hierarchical composition. For example, patterns can be used in different contexts to provide different information. For instance, the structural model specification is used to generate both the geometry of a character and the connections between geometry and animation models. One difficulty is the fact that not all aspects of a pattern or library are captured directly in its specification but may be provided by smaller-scale patterns. This requires knowledge or documentation about the pattern libraries being used. We have provided the POCL Browser as a means to provide this knowledge.

7.10.6 Reuse

Designing animation models for virtual characters is a difficult task. It is for this reason that we wish to promote the reuse of animation models and their components. We have built a comprehensive collection of patterns and pattern libraries representing various abstractions and abstraction layers that can be reused when constructing different ViB models.

Some of the specific components we have built can also be reused. Many of our motion controllers are general and can be reused between different characters. We have used the surface contact and surface friction controllers in several test examples. Action controllers, such as targeting and coordinating controllers, are also general and can be reused between characters with similar types of actions. Actions and low-level Modules, such as MoveForward, are typically specific to a particular geometry. However, because of structure of behavioral model, the higher-level modules, such as ApproachObject, can be reused between different characters, given that each character provide the required low-level Modules and Actions in a way specific to the character.

7.11 Summary

In this chapter, we presented an example character in order to illustrate how POCL can be used to construct a fairly complex ViB model consisting of several different

levels or layers. Because of the patterns provided in the previous chapter, the specification of the entities that make up the sub-models for each layer is done at a level of abstraction that is appropriate for each sub-model. For instance, the *ControllerModel* consists of multiple *ActionControllers* and a *PostureController*, the *ActionModel* consists of *Actions* arranged in *ActionLayers* and the *BehavioralModel* action-sequencing is made up of interconnected *Modules* containing *FSMs*.

The purpose of the lower-level layers in this architecture is to simplify higher-level ones. For instance, the *MotionModel* frees upper-level models, and, more particularly, the *ActionModel*, from needing to incorporate the effects of an *Action* on the global location of the character. The *PostureController* isolates higher-level behaviors from the underlying *MotionModel* and is key to ensuring proper low-level motion. The *ActionModel* organization ensures that certain *Actions* are only activated when appropriate, thus freeing the *BehavioralModel* from ensuring only appropriate activations occur.

Finally, several of the examples in this chapter have illustrated how library extension supports reuse. The *TargettingController*, *CycleController* and *ApproachObjectOverride* are example libraries that provide capabilities that can be reused through library extension. Specific examples of this include targeting controllers for the character's body, head and hands, cycle controllers for the character's body and head and approach object behaviors for approaching the TV and the fridge.

The resulting animation for our character, although not perfect, is acceptable. Considering the minimal low-level facilities provided by the underlying rendering architecture and, hence, the amount of animation control provided by the ViB model, we consider this to be good indication of the potential of this approach. The layered design of our model should allow us to upgrade the lower-level facilities while reusing the higher-level action and behavioral models with minimal to no modification. Because of the size of our model, run-time efficiency is an issue. We will discuss possible ways to improve this in the next chapter where we present our conclusions and possible avenues or future work.

Chapter 8

Conclusion

In this chapter, we will summarize our work and restate our contribution. We will also present some avenues of future work that could enhance our approach.

8.1 Summary

Our goal for this thesis was to specify and implement animation models for virtual characters and to promote the reuse of these models. In order to accomplish this goal, we considered two aspects of animation models, namely their representation and their specification. In Chapter 3, we presented the ViB model and supporting ViB engine. The key aspect of the ViB Engine is that the state of a ViB model is calculated as a difference from the previous state. Since the ViB model consists of a small number of low-level concepts used to define an animation model, the implementation of the supporting architecture and the ViB engine is straight forward as is the low-level interfaces that are used to connect the ViB engine to other architectures. The success of this model and its supporting architecture hinges on the ability to build animation models capable of providing complex behaviors for virtual characters.

Specification of animation models at the level of abstraction provided by the ViB model is not practical. In Chapter 4, we introduced pattern oriented composition as a means of specifying behaviors for use with the ViB engine using the higher-level abstractions that are commonly used in animation model creation. The focus of pattern

oriented composition is the hierarchical composition of the entities used to construct a model. By separating the low-level creation and manipulation of entities from the language itself, we were able to design a simple compiler and support environment in which to define patterns and pattern libraries. However, this separation relies on the inclusion of pattern primitives for both general composition and the creation of ViB model entities. For these particular domains, we have provided primitives that provide the lowest-level relationships possible so the number of pattern primitives required is small.

It is the abstractions built using these low-level primitives that aid in the specification of higher-level models. The ViB primitives are presented in Chapter 5 along with the high-level library organization used to support ViB model specification. These primitives allow families of variables with different characteristics to be defined, eliminating the need to provide built-in variable types. Because this is done directly with patterns, it is easy to add new types with new characteristics without impacting the existing higher-level facilities; i.e. the variable patterns are highly extensible.

Chapter 6 describes the AniMoS pattern libraries we created on top of the ViB and POCL primitive patterns in order to support animation model specification using the common animation model abstractions. These libraries provide patterns representing entities such as actions and modules, and also provide meta-level abstractions that define the relationships between these entities such as how they are interconnected or how their effects are combined or blended. The provided libraries are organized in such a way as to layer the animation model architecture. This not only allows the layers to be used separately but also allows particular layers and their sub-libraries to be overridden or modified using the library extension features provided by POCL.

Finally, Chapter 7 shows how we used the AniMoS pattern libraries to create a virtual character. Using these libraries, we built a virtual character whose behavior is completely controlled by a ViB model. This illustrates both the capabilities of the ViB engine and pattern oriented composition. The ViB model is capable of representing low-level motion that includes physical notions and, although the motion is no substitute for physics-based modelling, it is adequate for testing higher-level behaviors for articulated characters. However, for higher-level models such as action

control and action selection, the ViB model is capable of representing very complex models. As for pattern oriented composition, although the use of dynamic scoping makes pattern input determination more difficult, POCL does allow us to achieve our goal of building ViB models using a higher-level of abstraction. POCL would benefit from additional facilities as described in Section 8.3.

8.2 Contribution

The contribution of this thesis can be divided into three categories.

- The ViB model and its supporting ViB engine which is a differencing engine that provides a compact architecture that is capable of representing and evaluating animation models. The ViB model is capable of representing both kinematic and behavioral animation models and can be used to provide the complete animation model of a virtual character. Because of its simplicity, the ViB engine could easily be incorporated into or interfaced with other architectures in order to provide them with the capability of animating virtual characters.
- The POCL specification environment which incorporates a minimal number of variable scoping mechanisms combined with ideas from Alexander's patterns to create a low-level specification mechanism that can be used to build higher-level abstractions. By using this environment, we were able to create a comprehensive collection of patterns and pattern libraries representing the commonly used animation model abstractions and the relationships between them. Not only could these be used by others to create animation models for use with the ViB engine but they can also be overridden and tailored to meet the specific needs of a particular architecture.
- A layered virtual character architecture that represents a fusion of many of the commonly used animation model abstractions. The architecture layering scheme used is similar to that of Blumberg [11] and Perlin [54, 55] but is extended to provide coordination between the Behavioral and Action models through the Controller model. Each layer is composed of a set of entities with well defined

interfaces between them thus providing scalability within each layer. Abstraction barriers are provided between layers even though the layers are represented within a single ViB model. The controller and action layers act to isolate the behavioral model from the lower-level motion model and the geometric model. The division between controllers and actions acts to separate the control variables from their effects on the DOFs of the character. These arrangements provide the potential to replace lower-layers with other models such as a physics-based model while minimizing the impact on the higher layers.

To our knowledge, we are the first to use a single language that deals strictly with composition to generate animation models consisting of both lower-level layers that provide kinematic style animation and higher-level layers that provide behavioral animation. Furthermore, the resulting animation is reasonably natural looking and robust. The key to the success of this approach is the representation provided by the ViB model, the compositional facilities provided by POCL, and the pattern libraries that map the high-level specifications into this representation. The combination of these facilities allow use to overcome the shortcomings of previous animation architectures, as described in Section 2.2.7, by enabling different models to be used together within a single architecture and by providing a simple interface to other architectures.

8.3 Future Work

The following potential areas of future work also act to identify some of the shortcomings we encountered using the ViB and POCL environments.

- **Connectivity:** We have used the ViB engine within a single virtual character architecture. Although the ViB model is capable of providing multiple levels of behavior, the use of lower-level motion models such as physics-based modeling and inverse kinematics would enhance the animation produced using the engine. Furthermore, models that provide higher-level planning capabilities, such as cognitive models, along with realistic perception would aid in creating sophisticated autonomous creatures. Hence, it would be beneficial to interface

the ViB engine to architectures that provide these facilities. The ViB model could also potentially be used to provide behavioral models for robotics.

- **Portability:** Our implementation of the ViB engine is specific to the platform and environment we implemented it in. The next step is to implement the ViB engine as a virtual machine that can be used on multiple platforms and interfaced to different architectures to provide them with control of virtual characters. Included with this is the necessity to define an exchange format for ViB models. This would allow the ViB engine to be used to provide animation for VRML based scenes as done in [24].
- **Availability:** The patterns and pattern libraries used to create ViB models also need to be made generally available in order to promote the use of the ViB engine. The AniMoS libraries that we have created can be easily made available for others to use. Some of the general libraries we used to create specific types of behaviors such as targeting controllers could also be made available in separate libraries instead of being associated with the APE architecture. Furthermore, there are several other low-level and higher-level behavior model abstractions that could be implemented using POCL and made available for use.
- **Efficiency:** One of the main concerns with the ViB model is that, as the size of the model grows, so does the number of systems, channels and contribution functions that need to be evaluated on each tick. We have already addressed this issue to some degree by pruning excess channels and contributions functions from a model and by incorporating the ability to disable systems when possible. However, there are other potential means of increasing efficiency that we have not investigated. For instance, it may be possible to forward propagate channel output changes in order to identify and evaluate only those systems and channels that may potentially change. Another possible increase in execution speed could be to re-implement the ViB engine in C/C++.
- **Scalability:** Although the efficiency measures mentioned in the previous point do help improve scalability, they only help to push the boundary. Another potential means to improve scalability is to support the partitioning of a ViB model

into sub-models that could be evaluated on separate processors or processes. Support would be needed to either specify or identify the optimal partitioning so that the interactions between sub-model partitions is minimal.

- **Learning:** Once created, the structure of a ViB model is static. A means of dynamically modifying the structure could not only help address scalability but also allow for learned associations to be added to the model as was done in [11]. We have done some initial experiments with dynamically adding and removing channel contributions but more needs to be done to provide a framework that supports dynamic modification that represents learning. In addition, the ability to add whole channels and sub-systems could also be used to incorporate learning of new behaviors.
- **Specification Complexity:** From a specification standpoint, pattern oriented composition can be viewed as another style of programming. However, we have minimized the facilities that are provided in order to keep the supporting architecture small. Although this does allow access to multiple levels of abstraction within the same specification environment, it also leads to the creation of some pattern definitions that are not intuitive. Furthermore, the use of dynamic scoping puts the onus on the designer to find out both the explicit and implicit inputs of a pattern. Finally, the ability to reuse pattern libraries is greatly dependent on the library organization. Adding support to identify implicit inputs and promote proper library organization would greatly enhance POCL. Ideas from the area of domain specific languages [8] may help achieve this.
- **Compilation:** As the size of an animation model grows, so does the amount of time taken to compile it into the resulting ViB model. We have added extensions to the compiler to support the incremental compilation of each animation model layer but at the present moment these are still ad hoc and compiling large layers is still time consuming. Therefore, we need to formalize the incremental compilation abilities, integrate them with the POCL environment and extend them so they may be used for single entities such as actions or behavior modules. This capability will also help aid the partitioning and dynamic combination of

ViB models.

- **Reuse:** We have used POCL to support the reuse of pattern definitions and pattern libraries. Reuse could be extended by supporting the import of motions generated by other systems. Data from motion capture or other motion generating techniques could be translated to corresponding pattern definitions.
- **Tools:** We have provided a basic set of tools that aid in the creation and execution of ViB models. However, more sophisticated tools would further help this process. For instance, the ability to graphically create and relate patterns would be beneficial. Similarly, the ability to view the output of channels graphically during execution would aid in debugging animation models as would the ability to view a selected set of channels.

8.4 Conclusions

In this thesis, we have accomplished our goal of designing and implementing a framework that promotes virtual character animation model reuse. We accomplished this by defining the low-level ViB model representation and engine that is capable of representing multiple levels of behavior. We have also provided the POCL environment as a specification mechanism for allowing high-level animation model specifications to be translated to the low-level ViB model representation. Finally, we have shown how this architecture supports the fusion of many common but different behavior sub-architectures into one unified whole.

Bibliography

- [1] C. Alexander. *The Timeless Way of Building*. Oxford University Press, 1979. NA2500 A45.
- [2] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language*. Oxford University Press, 1977. HT166 A6147.
- [3] A.L. Ames, D.R. Nadeau, and J.L. Moreland. *VRML 2.0 Sourcebook*. Wiley, 1997.
- [4] R. Aylett. Virtual teletubbies: reapplying a robot architecture to virtual agents. In *Proceedings of the third annual conference on Autonomous Agents*, pages 338–339, 1999.
- [5] N. Badler, C. Phillips, and B. Webber. *Simulating Humans: Computer Graphics Animation and Control*. New York: Oxford Univ. Press, 1993. TA166.B32.
- [6] N.I. Badler, M.S. Palmer, and R. Bindiganavale. Animation control for real-time virtual humans. *Communications of the ACM*, 42(8):64–73, August 1999.
- [7] C. Balkenius. Natural intelligence for autonomous agents. In *Proceedings of the International Workshop on Mechatronical Computer Systems for Perception and Action*, pages 181–189. Hamstad University, 1993.
- [8] D. Batory, B. Lofaso, and Y. Smaragdakis. Jts: Tools for implementing domain-specific languages. In *5th International Conference on Software Reuse*, pages 143–53, 1998.

- [9] D. Bergen, J. Hahn, and P. Bock. An adaptive approach for reactive actor design. In *Proceedings of the Fourth European Conference on Artificial Life*, 1997.
- [10] B. Blumberg. Action-selection in hamsterdam: Lessons from ethology. In *Animals To Animats, Proceedings of the Third International Conference on the Simulation of Adaptive Behavior*. MIT Press, Cambridge MA, 1994.
- [11] B. Blumberg. *Old Tricks, New Dogs: Ethology and Interactive Creatures*. PhD thesis, MIT Media Laboratory, 1996.
- [12] B. Blumberg and T. Galyean. Mutli-level direction of autonomous creatures for real-time virtual environments. In *SIGGRAPH 95*, volume 29(3), pages 47–54, 1995.
- [13] R. Boulic. The humanoid environment for interactive animation of multiple deformable human characters. In *Procs. Eurographics 95*, pages 337–348, 1995.
- [14] R. Boulic, P. Becheiraz, L. Emering, and D. Thalmann. Integration of motion control techniques for virtual human and avatar real-time animation. In *Proceedings of the ACM International Symposium VRST'97*, pages 111–118, 1997.
- [15] R. Boulic, N. Magnenat-Thalmann, and D. Thalmann. A global human walking model with real-time kinematic personification. *The Visual Computer*, 6:244–358, October 1990.
- [16] D.C. Brogan, R.A. Metoyer, and J.K. Hodgins. Dynamically simulated characters in virtual environments. *IEEE Computer Graphics and Animation*, pages 58–69, September 1998.
- [17] R.A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.
- [18] A. Bruderlin and T. Calvert. Goal-directed, dynamic animation of human walking. *Journal of Computer and Graphics*, 23(3):233–242, July 1989.

- [19] A. Bruderlin, C. G. Teo, and T. Calvert. Procedural movement for articulated figure animation. *Computers and Graphics*, 18(4):453–461, 1994.
- [20] A. Bruderlin and L. Williams. Motion signal processing. In *SIGGRAPH 95*, volume 29(3), pages 97–104, 1995.
- [21] T. Calvert, A. Bruderlin, J. Dill, T. Schiphorst, and C. Welman. Desktop animation of multiple human figures. *IEEE Computer Graphics and Animation*, pages 18–26, May 1993.
- [22] J.J. Craig. *Introduction to Robotics*. Addison-Wesley, 1989.
- [23] D. Ebert, F. Musgrave, D. Peachey, K. Perlin, and S. Worley. *Texturing and Modeling: A Procedural Approach*. Academic Press, 1998.
- [24] N. Farenc, C. Babski, F. Garat, and D. Thalmann. Database and virtual human representation on the web. In *Advances in Databases and Multimedia for the New Century: A Swiss-Japanese Perspective*, 2000.
- [25] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics, Principles and Practice, Second Edition*. Addison-Wesley, 1990.
- [26] R.B. Fuller. *Synergetics: Explorations into the Geometry of Thinking*. Macmillan Publishing, 1979.
- [27] J. Funge. *Making Them Behave: Cognitive Models for Computer Animation*. PhD thesis, University of Toronto, 1998.
- [28] J. Funge, X. Tu, and D. Terzopoulos. Cognitive modeling: Knowledge, reasoning and planning for intelligent characters. In *SIGGRAPH 99*, pages 29–38, 1999.
- [29] M. Girard and A.A. Maciejewski. Computational modeling for the computer animation of legged figures. In *SIGGRAPH 85*, volume 19, pages 263–270, July.
- [30] M. Gleicher. Retargeting motion to new characters. In *SIGGRAPH 98*, pages 33–42, 1998.

- [31] P. Goetz. *Attractors in Recurrent Behavior Networks*. PhD thesis, University of New York, 1997.
- [32] J. Gomez. Twixt: A 3d animation system. *Computers and Graphics*, 9(3):291–298, 1985.
- [33] L. Gritz and J. Khan. Genetic programming for articulated figure motion. *Journal of Visualization and Computer Animation*, 6:129–142, 1995.
- [34] H. Haken. *Synergetics: An Introduction*. Springer-Verlag, 1977. Q295 H35.
- [35] J. Hodgins, W. Wooten, D. Brogan, and J. O'Brien. Animating human athletics. In *SIGGRAPH 95*, volume 19, August 1995.
- [36] M. B. Johnson. *WavesWorld: A Testbed for Three Dimensional Semi-Autonomous Animated Characters*. PhD thesis, MIT Media Laboratory, 1995.
- [37] P. Kalra, N. Magnenat-Thalmann, L. Moccozet, G. Sannier, A. Aubel, and D. Thalmann. Real-time animation of realistic virtual humans. *IEEE Computer Graphics and Animation*, pages 42–56, September 1998.
- [38] J.A.S. Kelso. *Dynamic Patterns: The Self-Organization of Brain and Behavior*. MIT Press, 1995. QP360 K454.
- [39] J.A.S. Kelso and A.S. Pandya. Dynamic pattern generation and recognition. In *Making Them Move; Mechanics, Control and Animation of Articulated Figures*, pages 171–190, 1991.
- [40] H. Ko and N.I. Badler. Animating human locomotion with inverse dynamics. *IEEE Computer Graphics and Animation*, pages 50–59, March 1996.
- [41] Y. Koga, K. Kondo, J. Kuffner, and J. Latombe. Planning motion with intentions. In *SIGGRAPH 94*, pages 395–408, July 1994.
- [42] J. Lee and S.Y. Shin. A hierarchical approach to interactive motion editing for human-like characters. In *SIGGRAPH 99*, pages 39–48, August 1999.

- [43] P. Maes. Situated agents can have goals. In *Designing Autonomous Agents*, pages 49–70. MIT Press, 1990.
- [44] N. Magnenat-Thalmann and D. Thalmann. *Synthetic Actors in Computer-Generated Films*. Springer-Verlag, Heidelberg, 1990.
- [45] N. Magnenat-Thalmann and D. Thalmann. Complex models for animating synthetic actors. *IEEE Computer Graphics and Animation*, 11(5):32–44, 1991.
- [46] M. Mataric, V. Zordon, and M. Williamson. Making complex articulated agents dance: an analysis of control methods drawn from robotics. In *Autonomous Agents and Multi-Agent Systems*, volume 2(1), pages 23–44, 1999.
- [47] Maxis. The sims. Computer Game, 2000.
- [48] S. F. May, W. Carlson, F. Phillips, and F. Scheepers. Al: A language for procedural modeling and animation. ACCAD OSU-ACCAD-12/96-TR5, The Ohio State University, December 1996.
- [49] G. Miller. The motion dynamics of snakes and worms. In *SIGGRAPH 88*, volume 22, pages 169–178, August 1988.
- [50] B. Muller and J. Reinhardt. *Neural Networks*. Springer Verlag, 1991.
- [51] J.T. Ngo and J. Marks. Spacetime constraints revisited. In *SIGGRAPH 93*, pages 343–350, 1993.
- [52] H. Noser and D. Thalmann. Sensor based synthetic actors in a tennis game simulation. *The Visual Computer*, 14(4):193–205, 1998.
- [53] I.S. Pandzic, T.S. Capin, E. Lee, N. Magnenat-Thalmann, and D. Thalmann. Autonomous actors in networked collaborative virtual environments. In *Proc. MultiMedia Modeling '98*, pages 138–145. IEEE Computer Society Press, 1998.
- [54] K. Perlin. Real time responsive animation with personality. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):5–15, March 1995.

- [55] K. Perlin and A. Goldberg. Improv: A system for scripting actors in virtual worlds. In *SIGGRAPH 96*, volume 30(3), pages 205–216, 1996.
- [56] C Phillips and N. Badler. Interactive behaviors for bipedal articulated figures. In *SIGGRAPH 91*, volume 25, pages 359–362, July 1991.
- [57] Z. Popovich and A. Witkin. Physically based motion transformation. In *SIGGRAPH 99*, pages 11–20, 1999.
- [58] W.T. Reeves. Particle systems-a technique for modeling a class of fuzzy objects. In *SIGGRAPH 83*, volume 17(3), pages 359–376, July 1983.
- [59] C.W. Reynolds. Computer animation with scripts and actors. volume 16(3), pages 289–296, July 1982.
- [60] C.W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *SIGGRAPH 87*, volume 21(4), pages 25–34, July 1987.
- [61] F. Scheepers, R. E. Parent, W. E. Carlson, and S. F. May. Anatomy-based modeling of the human musculature. In *SIGGRAPH 97*, pages 163–172, 1997.
- [62] G. Schoner and J.A.S. Kelso. Dynamic patterns of biological coordination: Theoretical strategy and new results. In *Dynamic Patterns in Complex Systems*, pages 77–102, 1988.
- [63] K. Shoemake. Animating rotation with quaternion curves. In *SIGGRAPH 85*, pages 143–152, 1985.
- [64] K. Sims. Evolving virtual characters. In *SIGGRAPH 94*, volume 28, pages 15–22, 1994.
- [65] G. Steele. *Common Lisp the Language, 2nd edition*. Digital Press, 1990.
- [66] D. Terzopoulos. Artificial life for computer graphics. *Communications of the ACM*, 42(8):33–42, August 1999.

- [67] D. Terzopoulos and K. Feischer. Modeling inelastic deformation: Viscoelasticity, plasticity, fracture. In *SIGGRAPH 88*, volume 22, pages 269–278, August 1988.
- [68] D. Terzopoulos, J. Platt, A. Barr, and K. Feischer. Elastically deformable models. In *SIGGRAPH 87*, volume 21(4), pages 205–214, July 1987.
- [69] D. Thalmann. Virtual sensors: A key tool for the artificial life of virtual actors. In *Proc. Pacific Graphics '95*, pages 22–40, 1995.
- [70] D. Thalmann. Physical, behavioral and sensor-based animation. In *Procs. Graphicon 96*, pages 214–221, 1996.
- [71] X. Tu. *Artificial Animals For Computer Animation: Biomechanics, Locomotion, Perception, Behavior*. PhD thesis, University of Toronto, 1996.
- [72] X. Tu and D. Terzopoulos. Artificial fishes: Physics, locomotion, perception, behavior. In *SIGGRAPH*, pages 43–50. ACM Press, July 1994.
- [73] R. Turner and D. Thalmann. The elastic surface layer model for animated character construction. In *Proc. Computer Graphics International '93*, pages 399–412. Springer-Verlag, 1993.
- [74] T. Tyrell. *Computational Mechanism for Action selection*. PhD thesis, Center for Cognitive Science, Edinburgh University, 1993.
- [75] M. Unuma, K. Anjyo, and R. Takeuchi. Fourier principles for emotion-based human figure animation. In *SIGGRAPH 95*, pages 91–96, 1995.
- [76] M.F. Valenta, D. Deugo, and W. LaLonde. Vibis: Specifying virtual behaviors as patterns. In *IC-AI'2000*, pages 603–609, July 2000.
- [77] M.F. Valenta and W. LaLonde. Motion composition and blending using motion patterns. In *Proceedings of the IASTED International Conference on Computer Graphics and Imaging*, pages 11–14. IASTED/ACTA Press, 1998.
- [78] M. van de Panne. Parameterized gait synthesis. *IEEE Computer Graphics and Animation*, pages 40–49, March 1996.

- [79] M. van de Panne and E. Fiume. Sensor-actuator networks. In *SIGGRAPH 93*, pages 335–342, 1993.
- [80] M. van de Panne, E. Fiume, and Z. Vranesic. Reusable motion synthesis using state-space controllers. In *SIGGRAPH 90*, volume 24, pages 225–234, August 1990.
- [81] J. Wilhelms. Using dynamic analysis for realistic animation of articulated bodies. *IEEE Computer Graphics and Animation*, pages 12–27, June 1987.
- [82] J. Wilhelms and R. Skinner. A notion for interactive behavior animation control. *IEEE Computer Graphics and Animation*, pages 14–22, May 1990. T385.A1I3.
- [83] A. Witkin and M. Kass. Spacetime constraints. In *SIGGRAPH 88*, volume 22, pages 159–168, 1988.
- [84] A. Witkin and Z. Popovic. Motion warping. In *SIGGRAPH 95*, pages 105–108, 1995.
- [85] D. Zelzer. Motor control techniques for figure animation. *IEEE Computer Graphics and Animation*, 2(9):53–59, November 1982.
- [86] D. Zelzer. Towards an integrated view of 3-d computer animation. *The Visual Computer*, 1(4):249–259, December 1985.

Appendix A

Miscellaneous Metrics

In this appendix, we describe the size of the implementation required to create the Robby character described in Section 7.

A.1 Smalltalk Implementation

The implementation of the underlying framework was done in Smalltalk. The following table gives the approximate number of object classes used in the implementation.

Component	Classes
ViB Engine	20
ViB Functions	40
POCL	25
APE	20
VRML	100

A.2 Primitive Patterns

Primitive patterns provide the translation of our pattern specifications into the underlying model representation. The following table gives the approximate number of primitives for each architectural model used for our example.

Component	Primitives
ViB	20
POCL	25
APE	10
VRML	35

A.3 Patterns and Libraries

We have divided the number of pattern libraries, definitions and lines of pattern specification into two categories. The AniMoS entry represents the abstractions we provided to map the high level APE specifications into the underlying models.

Area	Libraries	Definitions	Lines
AniMoS	50	550	4800
APE	90	725	13300

A.4 ViB Model

The following table gives the size of the resulting ViB model used for Robby.

channels	1500
systems	600

Appendix B

Detailed ViB Expansions

This appendix presents, in more detail, the libraries and expansions discussed in Chapter 5. Figures B.1 and B.2 contain the definitions of the ViB and APE libraries as presented in Chapter 5 that we will build upon.

```
Library ViB {  
  ViBModel = Primitive ...  
  ...  
  ViB = Pattern ViBModel {  
    contributions = Combine(model{...})  
  }  
  ...  
  Library KinematicDisplacementChannel {...}  
  Library ActionModel {  
    ActionModel = Pattern actions {}  
    Library Action {  
      Action = Pattern ...  
      ...  
    }  
  }  
  Library BehavioralModel {...}  
}
```

Figure B.1: The ViB Library

```

Library APE {
  ArticulatedFigure = Primitive ...
  APE = Pattern figure{}
  ...
  Library ArticulatedObject {...}
  Library Ship extends ArticulatedObject {...}}

```

Figure B.2: The APE Library

B.1 The ArticulatedObject Library

Figure B.3 contains the outline of the ArticulatedObject library. The ArticulatedObject entry pattern configures the APE ArticulatedFigure pattern in such a ways as to allow each of the sub-models to be provided by the GeometryModel, ConnectionModel and AnimationModel patterns. The AnimationModel is further divided into the MotionModel and ActionModel patterns.

```

Library ArticulatedObject {
  ArticulatedObject = Pattern ArticulatedFigure{
    name = objectName
    connectors = Pattern ViB2VRML{
      model = Pattern ConnectionModel {}
    }
    geometry = Pattern VRML{
      model = Pattern GeometryModel {}
    }
    animation = Pattern ViB{
      model = Pattern AnimationModel {}
    }
  }
  AnimationModel = Pattern [MotionModel{} ActionModel{}]
  GeometryModel = Pattern ...
  ConnectionModel = Pattern ...
  MotionModel = Pattern ...
}

```

Figure B.3: The ArticulatedObject Library

The MotionModel and ActionModel patterns represent sub-models of the animation layer and, therefore, can be provided as layers using pattern libraries as well.

The ActionModel library is contained in the ViB library. Like the ViB pattern definition above, the ActionModel pattern definition in the ActionModel library accepts as input a specification representing a set of actions, as shown in Figure B.1.

```

MotionModel = Pattern [
  InitialPositioning{
    initialOrientation = initialOrientation:=(zero)
    initialLocation = initialLocation:=(zero)}
  KinematicDisplacementChannel{
    channel = ["location"]
    acceleration = channelOutput(["acceleration"], 0)}
  ChannelContribution{
    channel = ["acceleration"]
    contribution = DeltaOperator("REPLACE")}]

```

Figure B.4: Articulated Object Motion Model

The MotionModel definition shown in Figure B.4 is contained in the ArticulatedObject library. The MotionModel is responsible for creating the portion of the ViB Model that determines the articulated object's position. The MotionModel in Figure B.4 uses KinematicDisplacementChannel to allow the location to be controlled using the output of the acceleration channel. The ChannelContribution configuration sets the delta operator of the acceleration channel to REPLACE using the DeltaOperator channel property primitive so that contributions to the acceleration channel will provide the value to be used for the acceleration instead of a delta to the previous acceleration.

The MotionModel also sets the initial position and orientation of the object using the InitialPositioning pattern. The initialLocation and initialOrientation bindings make use of a facility to provide a default value should one not be provided. The initialOrientation:=(zero) variable reference either resolves to the value bound to the initialOrientation variable in the resolution context or, if the variable is not bound, to the value bound to the zero variable. The variable zero is a data type independent zero value provided by the ViB engine. The InitialPositioning pattern, shown in Figure B.5, simply sets the initial output of the location and orientation channels

using the pattern primitive named `initial` to set the initial output channel property.

```

InitialPositioning = Pattern [
  ChannelContribution{
    channel = ["location"]
    contribution = initial(initialLocation)}
  ChannelContribution{
    channel = ["orientation"]
    contribution = initial(initialOrientation)}}]

```

Figure B.5: InitialPositioning Definition

B.1.1 The Ship Library

The `ArticulatedObject` library provides most of the patterns required to build a ViB model. As an example of a specific articulated object, we are going to use the `Ship` library that extends the `ArticulatedObject` library. The `Ship` library modifies the `MotionModel` and declares several action patterns including the `Thrust` pattern as shown in the outline of the `Ship` library in Figure B.6.

```

Library Ship extends ArticulatedObject {
  Ship = Pattern ArticulatedObject {...}
  MotionModel = Pattern [super.MotionModel ...]
  Thrust = Pattern Action ...
  ...
}

```

Figure B.6: Ship Library

To create a `Ship`, we need to make use of the `APE` library in order to gain access to the `Ship` library. This can be done as follows.

```

APE{
  figure = Pattern Ship {
    initialLocation = 0 @ 1 @ 0}}

```

Figure B.7: Resolution of APE Configuration

Figure B.7 shows the resolution frames that result from this configuration of the Ship library within the APE library. The top-level frame in the resolution context corresponds to the POCL library and, hence, contains the high-level libraries and the POCL pattern primitives. The APE library is used by providing it with a pattern representing the desired figure which in our case is a Ship. The APE configuration creates the APE library resolution frame and then resolves the “figure” pattern. When the Ship configuration bound to the variable “figure” is resolved, the initialLocation variable is bound in a new frame, along with a frame for the super-library of Ship (the ArticulatedObject library) and a frame for the Ship library. After the library

frames are created, the Ship pattern is resolved in this resolution context. The Ship pattern definition (summarized in Figure B.6 and presented in more detail in Figure B.8) consists of a configuration of the ArticulatedObject pattern.

```
Ship = Pattern ArticulatedObject{
  objectName = "Ship"
  damping = damping:=(0.2)
  accelerationRate = accelerationRate:=(2)
  turnRate = turnRate:=(0.5)
  actions = Pattern [Thrust{} TurnLeft{} TurnRight{}]}
```

Figure B.8: Ship Pattern Definition

The Ship pattern definition shown in Figure B.8 provides the ArticulatedObject pattern with a name and various other values that are used to tailor the resulting motion, as we will see below. Given this definition of the Ship pattern, the continuation of the expansion started at the end of Figure B.7 is shown in Figure B.9. This results in the creation of a frame containing the bindings from the ArticulatedObject configuration. The configuration of ArticulatedObject then expands to its definition from the ArticulatedObject library (see Figure B.3).

Figure B.9: Ship Pattern Resolution

Before the expansion of ArticulatedFigure continues, the pattern configurations

corresponding to the three sub-models, AnimationModel, GeometryModel and ConnectionsModel, must be resolved. Figure B.10 shows the beginning of the expansion involving the AnimationModel configuration. The ArticulatedObject definition provides the AnimationModel pattern as the “model” pattern of the ViB library. Figure B.10 illustrates that the provided “model” pattern is expanded to the AnimationModel pattern in the context provided by the ViB library and thus has access to the abstractions provided by this library.

Figure B.10: AnimationModel Pattern Resolution

The AnimationModel pattern consists of a configuration of the ActionModel and MotionModel patterns. ActionModel is a general pattern provided by the ViB library which references “actions” (see Figure B.1) that, in our example, is provided by the Ship definition as shown in Figure B.8. The details of the ActionModel expansion are discussed later. The ViB library does not contain a definition or library for MotionModel so the definition from the Ship library is used (see Figure B.6). However, since MotionModel is resolved in the context of ViB library, the MotionModel definition may contain references to the KinematicDisplacementChannel library and the ViB pattern primitives.

```

MotionModel = Pattern [
  super.MotionModel{}
  ChannelContribution{
    channel = ["location"]
    contribution = delta(channelOutput(["location"])) * negated(damping)}}]

```

MotionModel in Ship (see above) is an extension of MotionModel in ArticulatedObject. As is the case with object oriented programming languages, a means of referencing the overridden pattern definition is required. This capability is provided using the *super* qualifier. Because Ship extends ArticulatedObject, the “super.MotionModel” reference in the above MotionModel expands to the definition in ArticulatedObject shown in Figure B.4. The above MotionModel configures the inherited MotionModel and adds another contribution to the location channel that dampens the inertia. So, when there is no acceleration, the ship will slow down and stop. The damping value will determine how quickly this occurs.

The expansion of the MotionModel configuration illustrates library extension. Library extension has similar characteristics to inheritance in object-oriented programming languages in the sense that an extending library has access to the pattern definitions from the library it extends. However, except for the special reference qualifiers *super* and *outer*, the same variable scoping rules described in Section 5.1.4 apply during pattern resolution. It is how the resolution frames for a library extension hierarchy are created that dictates how inheritance works in conjunction with dynamic scoping. As in Section 5.1.4, the resolution of a library configuration results in the creation of a resolution frame containing the variable bindings from the configuration. However, before the frame containing the definitions from the resolved library is created, a frame for the extended library is created and this acts as the parent of the extending library’s resolution frame. For the Ship library, this is shown in Figure B.7 where the lowest frame contains the bindings from the Ship library and the second lowest frame contains those from Articulated object which Ship extends.

B.1.2 The Action Library

The Action library is contained in the ActionModel library as shown in Figure B.1. When the ActionModel configuration is resolved, the expansion shown in Figure B.11, which is a continuation of that in Figure B.10, results. The configuration of ActionModel results in the creation of the corresponding resolution frame containing the Action library defined in ActionModel, followed by the expansion of the ActionModel entry pattern which configures the actions pattern. The binding for actions is obtained from the frame in Figure B.9, which expands to a configuration of the Thrust, TurnLeft and TurnRight patterns. These are defined in the Ship library as specialized Actions. So it's convenient to consider the Action first before we consider these special actions.

Figure B.11: ActionModel Pattern Resolution

The Action Library defines a relationship between a set of effects and an activation signal. More specifically, it could define Action as follows (this is a simplification of our actual definition)

```

Action = Pattern effects{
  activation = channelOutput([actionName "active"], 0)
  ChannelContribution = Pattern outer.ChannelContribution{
    contribution = weigh(contribution, activation)}}

```

to ensure that contributions to a channel in the effects are weighed by the activation signal associated with the action. It achieves this by configuring "effects" in a context in which the ChannelContribution pattern is modified to weigh the contribution by an activation level. Even though ChannelContribution is bound to a pattern primitive, it is possible to override it because it is treated the same way as any other variable.

In the bottom frame of Figure B.10 ChannelContribution is bound to the primitive.

We cannot override ChannelContribution in the Action library using super because this would modify how all contributions made in the context of Action library were performed. We only want to override those used in the “effects” of an action. As we will see below, there aren’t any in our example but a more sophisticated action, such as that presented in Chapter 6, may have additional channels for controlling other aspects of the action. Instead, we use outer so that lookup begins in the parent frame of where the overriding ChannelContribution is bound, as shown in Figure B.12.

The Ship library contains three actions: Thrust, TurnLeft and TurnRight. Each of these actions is a specific instance of the above general action whose effects are triggered in response to an activation signal. The Thrust action contributes an acceleration that is calculated using the accelerationRate and the forward direction, which is a function of the current orientation of the ship.

```
Thrust = Pattern Action{
  actionName = "Thrust"
  effects = Pattern ChannelContribution{
    channel = ["acceleration"]
    contribution = directionForward{} * accelerationRate}}
```

```
directionForward = Pattern rotateVector(0 @ 0 @ 1, channelOutput(["orientation"]))
```

The Action pattern provides a context in which the ChannelContribution pattern weighs its contribution by an activation level. The actionName is used to identify the channel used to activate the action. The effects of the action are provided as a pattern which is configured in the context provided by the Action pattern. The expansion of the Thrust configuration is shown in Figure B.12. The key to this expansion is the expansion of “outer.ChannelContribution”. When this reference is expanded, lookup begins in the frame that is the parent of where the pattern was bound. Therefore, the lookup begins in the top frame in Figure B.12 and proceeds up to the ViB library frame in Figure B.10.

The TurnLeft pattern of Ship uses the Action pattern in a similar way as Thrust.

```

: Thrust {}
Action {
  actionName = "Thrust"
  effects = Pattern ChannelContribution {
    channel = "acceleration"
    contribution = directionForward {} * accelerationRate}}
  [actionName = "Thrust"
  effects = Pattern ChannelContribution {...}]
  effects {
    activation = channelOutput([actionName "active"], 0)
    ChannelContribution = outer.ChannelContribution {
      contribution = weigh(contribution, activation)}}
  [activation = channelOutput(["Thrust" "active"], 0)
  ChannelContribution = Pattern outer.ChannelContribution {...}]
  ChannelContribution {
    channel = "acceleration"
    contribution = directionForward {} * accelerationRate}
  [channel = "acceleration"
  contribution = (0 @ 0 @ 1) * 2]
  outer.ChannelContribution {
    contribution = contribution * activation}
  [contribution
  = weigh((0 @ 0 @ 1) * 2, channelOutput(["Thrust" "active"], 0))]

```

Figure B.12: Action Pattern Resolution

```

TurnLeft = Pattern Action{
  actionName = "TurnLeft"
  effects = Pattern ChannelContribution{
    channel = ["orientation"]
    contribution = turn(turnRate, directionUp{}})}}

```

TurnRight is the same as TurnLeft except that the turnRate is negated.

B.1.3 Resulting ViB Model

The ViB model that results from the example presented above is

$$\begin{aligned}
location_0 &= (0 \ 1 \ 0) \\
orientation_0 &= \emptyset \\
location_t &\stackrel{+}{=} \Delta location_{t-1} - 0.2\Delta location_{t-1} + acceleration_t \Delta t^2 / 2 \\
acceleration_t &= 2((0 \ 0 \ 1) \times orientation_{t-1}) \times Thrust.active_t \\
orientation_t &\stackrel{+}{=} rotation(0 \ 1 \ 0 \ 0.5\Delta t) \times TurnLeft.active_t + \\
&\quad rotation(0 \ 1 \ 0 \ -0.5\Delta t) \times TurnRight.active_t
\end{aligned}$$

The three active channels output either a 0 or 1 and are used to control the contributions to the acceleration and orientation channels. When the output of these channels is 0, the contribution to the corresponding channel is also 0 and when they are 1, the contribution function's output is applied to the channel. This allows the effects of either of the three actions to be activated and deactivated by the corresponding active channels. For this example, the active channels are external channels that are connected to boolean activation commands from the user issued through keystrokes.

For orientations, $\emptyset$ is the identity rotation and indicates the starting orientation for the ship. Also, the delta and blend operators for a channel that outputs orientations combine the rotations as if they were applied in sequence. This is not truly independent of order but for suitably small orientation deltas, it is acceptable. A rotation is specified using a 4-tuple containing the axis and angle of rotation. So for the rotations in the above model, turning is done about the y axis. A vector multiplied by a rotation, $(0 \ 0 \ 1) \times orientation_{t-1}$, results in a vector rotated by the rotation.

Appendix C

Pattern Libraries

C.1 POCL Library

```

Library POCL in () {
  ApplyBinaryOperator = Primitive signature(operand1 operator operand2) POCLBinaryOperation
  ApplyOperator = Primitive signature(operands operator) POCLOperation
  ApplyUnaryOperator = Primitive signature(operand operator) POCLUnaryOperation
  ArrayElement = Pattern signature(array, index) ApplyBinaryOperator(array, "at:", index)
  ArraySize = Pattern signature(array) getAttribute(array, "size")
  Choice = Primitive signature(condition ifTrue ifFalse) POCLChoicePattern
  Collect = Primitive signature(array pattern) POCLCollectPattern
  Combine = Primitive signature(children) POCLCombinePattern
  Concatenate = Primitive signature(strings makeName) POCLConcatenatePattern
  CopyArray = Primitive signature(array startingIndex endingIndex) POCLCopyArrayPattern
  CreateContext = Primitive signature(parameters values) POCLCreateContextPattern
  CreateContextArray = Primitive signature(parameters values fillStyles) POCLCreateContextArrayPattern
  ExtractEntities = Primitive signature(children) POCLExtractEntitiesPattern
  ExtractEntity = Primitive signature(children) POCLExtractEntityPattern
  false = POCLFalse
  FALSE = POCLFalse
  getAttribute = Pattern signature(entity, attribute) ApplyUnaryOperator(entity, attribute)
  Halt = Primitive HaltNode
  Inject = Primitive signature(seed array pattern) POCLInjectPattern
  Interval = Primitive signature(from to by) POCLIntervalPattern
  LocalVariableIdentifier = Pattern signature(variableName) variableName
  LocalVariableQualifier = Pattern LocalVariableIdentifier([])
  Parameter = Primitive signature(parameterName parameterDefault) POCLParameterReferencePattern
  PartialResolve = Primitive signature(pattern context) POCLPartialResolvePattern
  ProperNoun = Pattern signature(string) ApplyUnaryOperator(string, "asProperNoun")
  RemoveFirst = Pattern signature(array) CopyArray(array, 2, ArraySize(array))
  RemoveLast = Pattern signature(array) CopyArray(array, 1, ArraySize(array) - 1)
  Repeat = Primitive signature(array length) POCLRepeatPattern
  Resolve = Primitive signature(pattern context) POCLResolvePattern
  Select = Primitive signature(array pattern) POCLSelectPattern
  Stretch = Primitive signature(array length) POCLStretchPattern
  System = Pattern children{
    LocalVariableIdentifier = Pattern outer.LocalVariableIdentifier{
      variableName = outer.systemName + variableName}}
  TagForExtraction = Primitive signature(entity) POCLTagForExtractionPattern
  true = POCLTrue
  TRUE = POCLTrue
}

```


}

C.2 ViB Library

```

Library ViB in (POCL) {
  abs = Pattern signature(function) ApplyUnaryOperator(asSignal(function), "abs")
  AbsoluteChannelContribution = Pattern ChannelContribution{
    contribution = channelOutputDeltaTo(contribution)}
  and = Primitive signature(conditions) AndFunction
  angleBetweenDirections = Pattern signature(direction1, direction2) arcCos(
    maximum(minimum(dot(direction1, direction2), 1), 0))
  arcCos = Pattern signature(function) ApplyUnaryOperator(asSignal(function), "arcCos")
  asSignal = Primitive signature(function) Signal
  bias = Pattern signature(x, b) pow(x, log(b) / log(0.5))
  bitAnd = Pattern signature(function1, function2) ApplyBinaryOperator(function1, "bitAnd:", function2)
  Blend = Pattern Combine()
  BlendOperator = Primitive signature(operator) BlendOperator
  ChannelComparisonBlendOperator = Primitive signature(channel comparison) ChannelComparisonBlendOperator
  ChannelContribution = Primitive signature(channel contribution) ViBChannelContribution
  ChannelMetaContribution = Primitive signature(channel contribution) ViBChannelContribution
  channelOutput = Primitive signature(channel offset) ViBChannelOutput
  channelOutputDeltaTo = Pattern signature(function)
    ApplyUnaryOperator(asSignal(function), "asChannelOutputDeltaSignal")
  clamp = Pattern signature(function, min, max) maximum(
    minimum(asSignal(function), asSignal(max)), asSignal(min))
  clampChannelOutput = Pattern signature(min, max) ApplyBinaryOperator(
    asSignal(min), "asClampChannelSignal:", asSignal(max))
  ClampOutputOfChannel = Pattern ChannelMetaContribution{
    contribution = clampChannelOutput(minimumOutput, maximumOutput)}
  cos = Pattern signature(function) ApplyUnaryOperator(asSignal(function), "cos")
  cross = Pattern signature(vector1, vector2) ApplyBinaryOperator(
    asSignal(vector1), "crossProduct:", asSignal(vector2))
  delta = Pattern signature(function) ApplyUnaryOperator(asSignal(function), "asDeltaSignal")
  DeltaOperator = Primitive signature(operator) DeltaOperator
  derivative = Pattern signature(function) delta(asSignal(function)) / delta(asSignal(t))
  derivative2 = Pattern signature(function) (delta(asSignal(function))
    - delta(previousOutput(asSignal(function)))) / square(delta(asSignal(t)))
  directionTowards = Pattern signature(fromPoint, toPoint) ApplyBinaryOperator(
    asSignal(fromPoint), "directionTo:", asSignal(toPoint))
  distanceBetween = Pattern signature(fromPoint, toPoint) ApplyBinaryOperator(
    asSignal(fromPoint), "distanceFrom:", asSignal(toPoint))
  distanceSquaredBetween = Pattern signature(fromPoint, toPoint) ApplyBinaryOperator(
    asSignal(fromPoint), "distanceSquaredFrom:", asSignal(toPoint))
  dot = Pattern signature(vector1, vector2) ApplyBinaryOperator(
    asSignal(vector1), "dotProduct:", asSignal(vector2))
  epsilon = Pattern 0.1e-8
  equalDirectionsOnPlane = Pattern signature(direction1, direction2, normal) equalPoints(
    normalize(projectPointToPlaneThroughOrigin(asSignal(direction1), asSignal(normal))),
    normalize(projectPointToPlaneThroughOrigin(asSignal(direction2), asSignal(normal))))
  equalPoints = Pattern signature(point1, point2)
    magnitude(asSignal(point2) - asSignal(point1)) <= asSignal(equalityThreshold:=(epsilon))
  equalPointsOnPlane = Pattern signature(point1, point2, normal) equalPoints(
    projectPointToPlaneThroughOrigin(asSignal(point1), asSignal(normal)),
    projectPointToPlaneThroughOrigin(asSignal(point2), asSignal(normal)))
  externalChannelOutput = Primitive signature(system channel offset) ExternalChannelReference
  externalSystem = Primitive signature(systemName) ExternalSystemReference
  gain = Pattern signature(x, g) if(x < 0.5, bias(2 * x, 1 - g) / 2, (1 - bias(2 - (2 * x), 1 - g)) / 2)

```

```

if = Primitive signature(condition whenTrue whenFalse) ConditionalFunction
initial = Pattern signature(value) ApplyUnaryOperator(asSignal(value), "asInitialValueSignal")
integral = Pattern signature(function) ((asSignal(function) + previousOutput(function)) / 2) * delta(
    asSignal(t))
inverse = Pattern signature(value) ApplyUnaryOperator(asSignal(value), "inverse")
Literal = Primitive signature(literal) ViBLiteral
ln = Pattern signature(function) ApplyUnaryOperator(asSignal(function), "ln")
log = Pattern signature(function) ApplyUnaryOperator(asSignal(function), "ln")
magnitude = Pattern signature(vector) ApplyUnaryOperator(asSignal(vector), "magnitude")
maximum = Pattern signature(function1, function2) ApplyBinaryOperator(
    asSignal(function1), "max:", asSignal(function2))
minimum = Pattern signature(function1, function2) ApplyBinaryOperator(
    asSignal(function1), "min:", asSignal(function2))
mod = Pattern signature(function, modulus) ApplyBinaryOperator(
    asSignal(function), "mod:", asSignal(modulus))
negated = Pattern signature(function) ApplyUnaryOperator(asSignal(function), "negated")
normalize = Pattern signature(vector) ApplyUnaryOperator(asSignal(vector), "normalized")
not = Pattern signature(condition) ApplyUnaryOperator(asSignal(condition), "not")
number = Pattern signature(boolean) asSignal(boolean)
or = Primitive signature(conditions) OrFunction
pi = Pattern 3.14159265
pow = Pattern signature(function1, function2) ApplyBinaryOperator(
    asSignal(function1), "raisedTo:", asSignal(function2))
projectPointToPlane = Pattern signature(point, planePoint, planeNormal) asSignal(point)
    + (asSignal(planeNormal) * dot(planePoint - point, planeNormal))
projectPointToPlaneThroughOrigin = Pattern signature(point, normal) asSignal(point)
    - (asSignal(normal) * dot(point, normal))
random = Pattern signature(start, end) ApplyBinaryOperator(
    asSignal(start:=0), "randomTo:", asSignal(end:=1))
rotate = Pattern signature(value, rotation) ApplyBinaryOperator(
    asSignal(rotation), "rotate:", asSignal(value))
SetChannelToAndContributions = Pattern {
    ChannelMetaContribution{
        contribution = BlendOperator("and")}
    ChannelMetaContribution{
        contribution = DeltaOperator("replace:")}
}
SetChannelToMultiplyContributions = Pattern ChannelMetaContribution{
    contribution = BlendOperator("and")}
SetChannelToOrContributions = Pattern {
    ChannelMetaContribution{contribution = BlendOperator("or")}
    ChannelMetaContribution{contribution = DeltaOperator("replace:")}
}
SetChannelToOutputMaximumContribution = Pattern {
    ChannelMetaContribution{contribution = BlendOperator("max:")}
    ChannelMetaContribution{contribution = DeltaOperator("replace:")}
}
SetChannelToReplaceOutput = Pattern ChannelMetaContribution{
    contribution = DeltaOperator("replace:")}
SetInitialOutputOfChannel = Pattern ChannelMetaContribution{
    contribution = initial(output:=(initialOutput))}
SetMinimumOutputOfChannel = Pattern ChannelMetaContribution{
    contribution = clampChannelOutput(minimumOutput, 1000.0)}
sign = Pattern signature(function) ApplyUnaryOperator(asSignal(function), "sign")
sin = Pattern signature(function) ApplyUnaryOperator(asSignal(function), "sin")
square = Pattern signature(function) asSignal(function * function)
subsystem = Primitive signature(function systemName) SubsystemSignal
system = Primitive signature(systemName) SystemReference
t = Pattern channelOutput{
    LocalVariableIdentifier = Pattern signature(variableName) variableName
    channel = ["t"]
    offset = 0}-

```

```

turn = Pattern signature(velocity, axis) ApplyBinaryOperator(asSignal(
    velocity(t)) * asSignal(delta(t)), "asRotationAngleAround:", axis(t))
turnTo = Pattern signature(angle, axis) ApplyBinaryOperator(
    delta(angle(t)), "asRotationAngleAround:", axis(t))
ViB = Pattern signature(name, contributions) ViBModel(Combine(contributions{}))
ViBEngine = Primitive signature(name models) ViBEngine
ViBModel = Primitive signature(contributions) ViBModelEntity
weigh = Pattern signature(function, weight) ApplyBinaryOperator(function, "weight:", weight)
zero = Primitive signature(t) Zero
ZeroOnZeroDivide = Pattern signature(numerator, denominator)
    if(abs(denominator) < epsilon, zero, numerator / denominator)
}

```

C.3 AniMoS Library

```

Library AniMoS extends ViB in (POCL) {
    ActionInterfaceContext = Pattern Context {
        ActivateAction = LocalOperation{
            operation = Pattern Resolve{
                context = ActionModel
                pattern = Pattern ActivateAction{}}
            arguments = ["actionName"]}
        actionActivated = LocalOperation{
            operation = Pattern Resolve{
                context = ActionModel
                pattern = Pattern actionActivated{}}
            arguments = ["actionName"]}
        actionDeactivated = LocalOperation{
            operation = Pattern Resolve{
                context = ActionModel
                pattern = Pattern actionDeactivated{}}
            arguments = ["actionName"]}
        ChannelContribution = Pattern super.ChannelContribution{channel = LocalVariableIdentifier(channel)}
        ChannelMetaContribution = Pattern super.ChannelMetaContribution{channel = LocalVariableIdentifier(channel)}
        channelOutput = Pattern super.channelOutput{channel = LocalVariableIdentifier(channel)}
    }
}

```

```

ControllerInterfaceContext = Pattern Context {
  EnableController = LocalOperation{
    operation = Pattern Resolve{
      context = ControllerModel
      pattern = Pattern EnableController{}}
    arguments = ["controllerName"]}
  RequestPosture = LocalOperation{
    operation = Pattern Resolve{
      context = ControllerModel
      pattern = Pattern RequestPosture{}}
    arguments = ["postureName"]}
  currentPostureIs = LocalOperation{
    operation = Pattern Resolve{
      context = ControllerModel
      pattern = Pattern currentPostureIs{}}
    arguments = ["postureName"]}
  GetControllerVariable = LocalOperation{
    operation = Pattern Resolve{
      context = ControllerModel
      pattern = Pattern GetControllerVariable{}}
    arguments = ["controllerName" "variableName" "offset"]}
  SetControllerVariable = LocalOperation{
    operation = Pattern Resolve{
      context = ControllerModel
      pattern = Pattern SetControllerVariable{}}
    arguments = ["controllerName" "variableName" "value"]}
  DecreasingScalarReleasingMechanism = Pattern maxValue:=(1) - IncreasingScalarReleasingMechanism{
  EnableModule = Pattern signature(moduleName) ChannelContribution{
    channel = moduleName + ["enabled"]
    contribution = activation{}}
  EqualDirectionsReleasingMechanism = Pattern signature(direction1, direction2) DecreasingScalarReleasingMechanism{
    value = distanceSquaredBetween(direction1, direction2)
    activeRange = [0 square(threshold:=(0.2))]}
  EqualPointsOnPlaneReleasingMechanism = Pattern signature(point1, point2, normal) EqualPointsReleasingMechanism{
    projectPointToPlaneThroughOrigin(point1, normal),
    projectPointToPlaneThroughOrigin(point2, normal)}
  EqualPointsReleasingMechanism = Pattern signature(point1, point2) DecreasingScalarReleasingMechanism{
    maxValue = maxValue:=(1.0)
    activeRange = [epsilon * -1 square(equalityThreshold:=(epsilon))]
    value = distanceSquaredBetween(point2, point1)}
  GetModuleOutput = Pattern signature(moduleName, outputName) Resolve{
    context = Module
    pattern = Pattern GetModuleOutput{module = Parameter(moduleName)}}
  GetModuleOutputActivation = Pattern signature(moduleName, outputName) Resolve{
    context = Module
    pattern = Pattern GetModuleOutputActivation{module = Parameter(moduleName)}}
  IdentifierQualifier = Pattern LocalVariableQualifier{}
  IncreasingScalarReleasingMechanism = Pattern Resolve{
    pattern = Pattern minimum(1, maximum(value - start, 0.0) / (end - start)) * maxValue:=(1)
    context = CreateContext{
      parameters = ["start" "end"]
      values = activeRange}}
  LocalizeSystemReferencing = Pattern signature(children) children{
    LocalVariableIdentifier = Pattern signature(variableName) variableName}

```

```

LocalOperation = Pattern signature(operationContext, operation, arguments) PartialResolve{
  context = Context {
    operationContext = operationContext:=( Context {} )
    operation = operation
    operationQualifier = LocalVariableQualifier{ }
    patternSignature = arguments:={ }
    pattern = Pattern Resolve{
      context = operationContext
      LocalVariableIdentifier = Pattern signature(variableName) operationQualifier + variableName
      pattern = Pattern operation{ }
    }
  }
SetModuleInput = Pattern signature(moduleName, inputName, value) Resolve{
  context = Module
  pattern = Pattern ToModuleInput{ module = Parameter(moduleName) }
}
VariableDefinition = Pattern signature(variableSystem, variableName) Resolve{
  context = VariableSystem
  pattern = Pattern VariableDefinition(variableSystem, variableName)
}
VariableOperation = Pattern signature(variable, operation) variable(operation)
AniMoS = Pattern signature(name, contributions) ViB(name, contributions)
}

```

C.3.1 ActivatedChannelContribution Library

```

Library ActivatedChannelContribution in (POCL AniMoS) {
  ActivatedChannelContribution = Pattern children{ }
  ChannelContribution = Pattern [
    UnadjustedChannelContribution{ contribution = adjustContribution(activation, contribution) }
    ContributeToActivation{ channel = activationChannelIdentifier{ } }
  ]
  activationChannelIdentifier = Pattern signature(channel) channel + [ "activation" ]
  adjustContribution = Pattern signature(activation, contribution) weigh(contribution, activation)
  ChannelOutputWhenInactive = Pattern [
    UnadjustedChannelContribution{
      contribution = weigh(channelOutputDeltaTo(output), 1 - channelOutput(activationChannelIdentifier{ }, 0))
    }
    SetInitialOutputOfChannel{
      channel = activationChannelIdentifier{ }
      output = zero
    }
  ]
  ContributeToActivation = Pattern [
    UnadjustedChannelContribution{ contribution = activation }
    ClampOutputOfChannel{ maximumOutput = 1.0 minimumOutput = 0.0 }
    SetChannelToOutputMaximumContribution{ }
  ]
  UnadjustedChannelContribution = Pattern outer.ChannelContribution{ }
}

```

C.3.2 WeightedChannelContribution Library

```

Library WeightedChannelContribution extends ActivatedChannelContribution in (POCL AniMoS) {
  WeightedChannelContribution = Pattern ActivatedChannelContribution{ }
  ChannelContribution = Pattern super.ChannelContribution{ activation = activation:=(weight) }
  adjustContribution = Pattern signature(activation, contribution) weigh(contribution, weight)
}

```

C.3.3 NormalizedChannelContribution Library

```

Library NormalizedChannelContribution extends WeightedChannelContribution in (POCL AniMoS) {
  NormalizedChannelContribution = Pattern ActivatedChannelContribution{ }
  ChannelContribution = Pattern [
    super.ChannelContribution{
      activation = weight
      weight = weigh(weight, weightScaleFactor{ })
    }
    ContributeToTotalWeight{ channel = totalWeightChannelIdentifier{ } }
  ]
}

```

```

ContributeToTotalWeight = Pattern [
    UnadjustedChannelContribution{contribution = weight}
    SetChannelToReplaceOutput{}}
totalWeightChannelIdentifier = Pattern signature(channel).channel + totalWeightChannel:={"totalWeight"}
weightScaleFactor = Pattern ZeroOnZeroDivide(weight, channelOutput(totalWeightChannelIdentifier{ }, 0))
}

```

C.3.4 StrengthAdjustedChannelContribution Library

```

Library StrengthAdjustedChannelContribution extends NormalizedChannelContribution in (POCL AniMoS) {
    StrengthAdjustedChannelContribution = Pattern ActivatedChannelContribution{
    ContributeToTotalWeight = Pattern super.ContributeToTotalWeight{weight = weigh(strength:=(1), weight)}
    StrengthAdjustedChannelContribution = Pattern ActivatedChannelContribution{
    weightScaleFactor = Pattern super.weightScaleFactor{weight = weigh(strength:=(1), weight)}
}

```

C.3.5 ImportanceAdjustedChannelContribution Library

```

Library ImportanceAdjustedChannelContribution extends StrengthAdjustedChannelContribution in (POCL AniMoS) {
    ImportanceAdjustedChannelContribution = Pattern ActivatedChannelContribution{
    ActivatedChannelContribution = Pattern super.ActivatedChannelContribution{
        strength = weigh(strength:=(1), importance:=(1))
    }
    ImportanceAdjustedChannelContribution = Pattern ActivatedChannelContribution{
}

```

C.3.6 ChannelVariable Library

```

Library ChannelVariable in (POCL AniMoS) {
    ChannelVariable = Pattern DefaultEntryPattern{
    VariableDefinitionContext = Pattern Context {
        variableName = variableName
        variableQualifier = IdentifierQualifier{
        variableType = Pattern ChannelVariable}
    DefaultEntryPattern = Pattern Closure{
        context = VariableDefinitionContext{
        pattern = Pattern Resolve{
            context = variableType{
            pattern = operation}
        patternSignature = ["operation"]}
    CustomGetActivationPatternName = Pattern Concatenate([variableName "Activation"])
    CustomGetPatternName = Pattern variableName
    CustomizedOperations = Pattern signature(variable) Resolve{
        context = variable
        pattern = Pattern CreateContext{
            parameters = [CustomGetPatternName{ } CustomGetActivationPatternName{ }
                CustomSetPatternName{ } CustomSetDefaultPatternName{ }]
            values = [
                Pattern signature(offset) VariableOperation(outer.variable, Pattern GetVariable{ })
                Pattern signature(offset) VariableOperation(outer.variable, Pattern GetVariableActivation{ })
                Pattern signature(value) VariableOperation(outer.variable, Pattern SetVariable{ })
                Pattern signature(defaultValue) VariableOperation(outer.variable, Pattern SetVariableDefault{ })
            ]
        }
    CustomSetDefaultPatternName = Pattern Concatenate(["Set" variableName "Default"], true)
    CustomSetPatternName = Pattern Concatenate(["Set" variableName], true)
    GetVariable = Pattern signature(variableName, offset) channelOutput(variableName)
    GetVariableActivation = Pattern 1
    InitializeVariable = Pattern signature(variableName) []
    LocalVariableIdentifier = Pattern signature(variableName) variableQualifier + variableName
}

```

```

SetVariable = Pattern signature(variableName, value) ChannelContribution{
    channel = variableName
    contribution = value}
SetVariableDefault = Pattern SetInitialOutputOfChannel{
    channel = variableName
    initialOutput = defaultValue:=(value)}
}

```

C.3.7 ArticulatedVariable Library

```

Library ArticulatedVariable extends ChannelVariable in (POCL AniMoS) {
    ArticulatedVariable = Pattern DefaultEntryPattern{}
    DefaultCombinationScheme = Pattern ActivatedChannelContribution
    GetVariable = Pattern signature(variableName) Resolve{
        context = combinationScheme:=(DefaultCombinationScheme{})
        pattern = Pattern channelOutput(variableName)}
    GetVariableActivation = Pattern signature(variableName) Resolve{
        context = combinationScheme:=(DefaultCombinationScheme{})
        pattern = Pattern channelOutput(activationChannelIdentifier(variableName))}
    InitializeVariable = Pattern signature(variableName) Resolve{
        context = combinationScheme:=(DefaultCombinationScheme{})
        pattern = Pattern SetInitialOutputOfChannel{
            channel = variableName
            initialOutput = initialValue:=(zero)}}
    SetVariable = Pattern signature(variableName, value) Resolve{
        context = combinationScheme:=(DefaultCombinationScheme{})
        pattern = Pattern ChannelContribution{
            activation = activation(0)
            channel = variableName
            contribution = value}}
    SetVariableDefault = Pattern signature(variableName, defaultValue) Resolve{
        context = combinationScheme:=(DefaultCombinationScheme{})
        pattern = Pattern ChannelOutputWhenInactive{
            channel = variableName
            output = defaultValue}}
    VariableDefinitionContext = Pattern Context {
        variableName = variableName
        variableQualifier = IdentifierQualifier{}
        variableType = Pattern Variable
        combinationScheme = combinationScheme:=(DefaultCombinationScheme{})
        initialValue = initialValue:=(zero)}
}

```

C.3.8 Variable Library

```

Library Variable extends ArticulatedVariable in (POCL AniMoS) {
    InitializeVariable = Pattern signature(variableName) {
        super.InitializeVariable{}
        SetChannelToReplaceOutput{channel = variableName}}
}

```

C.3.9 MemoryVariable Library

```

Library MemoryVariable extends Variable in (POCL AniMoS) {
    InitializeVariable = Pattern {
        super.InitializeVariable{}
        SetChannelToReplaceOutput{channel = variableName}
        SetVariableDefault{defaultValue = GetVariable{}}}
}

```

```

MemoryVariable = Pattern DefaultEntryPattern{}
SetVariable = Pattern super.SetVariable(activation = activation() > 0)
VariableDefinitionContext = Pattern Context {
    variableName = variableName
    variableQualifier = IdentifierQualifier{}
    variableType = Pattern MemoryVariable
    initialValue = initialValue:=(zero)}
}

```

C.3.10 BlendedVariable Library

```

Library BlendedVariable extends Variable in (POCL AniMoS) {
    BlendedVariable = Pattern DefaultEntryPattern{}
    DefaultChannelContributionScheme = Pattern NormalizedChannelContribution
    SetVariable = Pattern super.SetVariable(weight = activation)
    VariableDefinitionContext = Pattern signature(variableName) Context {
        variableName = variableName
        variableType = Pattern BlendedVariable
        variableQualifier = IdentifierQualifier{}
        ContributionScheme = blendingScheme:=(DefaultChannelContributionScheme{})}
}

```

C.3.11 EnablingVariable Library

```

Library EnablingVariable extends ChannelVariable in (POCL AniMoS) {
    CustomizedOperations = Pattern Combine([super.CustomizedOperations{} Context {
        Enable = Pattern signature(activation) VariableOperation(variable, Pattern Enable(activation))
        Disable = Pattern signature(activation) VariableOperation(variable, Pattern Disable(activation))}])
    Disable = Pattern signature(activation) ChannelContribution{
        channel = "disable"
        contribution = activation}
    Enable = Pattern signature(activation) SetVariable(variableName, activation)
    enabled = Pattern signature(offset) GetVariable(variableName, offset)
    EnabledAlways = Pattern ChannelContribution{
        channel = ["enabled"]
        contribution = 1}
    EnabledUntilDisabled = Pattern [
        ChannelContribution{
            channel = ["enabled"]
            contribution = and([channelOutput(["enabled"])not(channelOutput(["disable"], 0))])
        }
        SetChannelToReplaceOutput{channel = ["disable"]}
        ClampOutputOfChannel{
            channel = ["disable"]
            minimumOutput = 0
            maximumOutput = 1}]
    EnabledWhileActivated = Pattern []
    EnablingVariable = Pattern DefaultEntryPattern{}
    InitializeVariable = Pattern [
        SetChannelToOrContributions{channel = ["enabled"]}
        SetInitialOutputOfChannel{
            channel = ["enabled"]
            initialOutput = zero}
        enablingScheme:{}]
    VariableDefinitionContext = Pattern Context {
        variableName = "enabled"
        variableQualifier = IdentifierQualifier{}
        variableType = Pattern EnablingVariable
        enablingScheme = enablingScheme:=(EnabledWhileActivated)}
}

```


C.3.12 SuppressibleVariable Library

```

Library SuppressibleVariable extends MemoryVariable in (POCL AniMoS) {
  CustomArrivalPatternName = Pattern Concatenate([variableName "Arrived"])
  CustomizedOperations = Pattern Combine([super.CustomizedOperations{
    CreateContext{
      parameters = [CustomArrivalPatternName{}]
      values = [Pattern signature(offset) VariableOperation(variable, Pattern GetVariableActivation()))]]
  InitializeVariable = Pattern [
    super.InitializeVariable{
      SetChannelToReplaceOutput{channel = variableName + ["suppressed"]}
    SetVariable = Pattern super.SetVariable{activation = and([activation() > 0 not(variableSuppressed{}))]}
    SetVariableSuppression = Pattern ChannelContribution{
      channel = variableName + ["suppressed"]
      contribution = activation() > 0}
    SuppressibleVariable = Pattern DefaultEntryPattern{
      variableSuppressed = Pattern channelOutput(variableName + ["suppressed"], 0)
    VariableDefinitionContext = Pattern Combine([
      super.VariableDefinitionContext{} Context {
        variableType = Pattern SuppressibleVariable{}}
  ]
}

```

C.3.13 InhibitableVariable Library

```

Library InhibitableVariable extends MemoryVariable in (POCL AniMoS) {
  InhibitableVariable = Pattern DefaultEntryPattern{}
  InhibitVariable = Pattern ChannelContribution{
    channel = variableName + ["inhibited"]
    contribution = activation() > 0}
  InitializeVariable = Pattern [
    super.InitializeVariable{
      SetChannelToReplaceOutput{channel = variableName + ["inhibited"]}
    SetVariable = Pattern super.SetVariable{activation = and([activation() > 0 not(variableInhibited{}))]}
    variableInhibited = Pattern channelOutput(variableName + ["inhibited"], 0)
  VariableDefinitionContext = Pattern Combine([
    super.VariableDefinitionContext{} Context {
      variableType = Pattern InhibitableVariable{}}
  ]
}

```

C.3.14 ActivationVariable Library

```

Library ActivationVariable extends Variable in (POCL AniMoS) {
  ActivationVariable = Pattern DefaultEntryPattern{}
  GetVariableActivation = Pattern GetVariable{}
  InitializeVariable = Pattern [
    super.InitializeVariable{
      SetChannelToOutputMaximumContribution{channel = variableName}
      ClampOutputOfChannel{
        minimumOutput = 0.0
        channel = variableName
        maximumOutput = 1.0}
    SetVariable = Pattern super.SetVariable{
      activation = 1
      value = value:=(activation(0))}
  VariableDefinitionContext = Pattern Context {
    variableName = variableName
    variableQualifier = IdentifierQualifier{}
    variableType = Pattern ActivationVariable}
}

```

```
}

```

C.3.15 DisplacementChannel Library

```
Library DisplacementChannel in (POCL AniMoS) {
  accelerate = Pattern signature(acceleration) (acceleration * square(delta(t))) / 2
  ApplyAccelerationToChannel = Pattern ContributeToAcceleration{
    channelDelta = Pattern signature(channel) delta(channelOutput(channel, 0))
    channelDeltaFeedback = Pattern signature(channel) delta(channelOutput(channel, -1))
    channelVelocity = Pattern signature(channel) channelDelta(channel) / delta(t)
    channelVelocityFeedback = Pattern signature(channel) channelDeltaFeedback(channel) / delta(t)
    ContributeLiftToAcceleration = Pattern ContributeToAcceleration{
      contribution = ((upDirection * dot(forwardDirection, channelVelocityFeedback(locationChannel))) * lift) * -1}
    ContributeResistanceToAcceleration = Pattern ContributeToAcceleration{
      contribution = (channelVelocityFeedback(channel) * damping) * -1}
    ContributeToAcceleration = Pattern ChannelContribution{
      contribution = accelerate(acceleration)}
    DisplacementChannel = Pattern [FeedbackInertiaToChannel{} ApplyAccelerationToChannel{}]
    FeedbackInertiaToChannel = Pattern ChannelContribution{contribution = delta(channelOutput())}
    move = Pattern signature(velocity) velocity * delta(t)
}
```

C.3.16 DeltaDisplacementChannel Library

```
Library DeltaDisplacementChannel extends DisplacementChannel in (POCL AniMoS) {
  ApplyAccelerationToChannel = Pattern ChannelContribution{
    channel = channel + ["delta"]
    contribution = accelerate(channelOutput(channel + ["acceleration"], 0))}
  channelVelocityFeedback = Pattern signature(channel) channelOutput(channel + ["velocity"])
  ContributeToAcceleration = Pattern ChannelContribution{channel = channel + ["acceleration"]}
  DeltaDisplacementChannel = Pattern DisplacementChannel{}
  DisplacementChannel = Pattern [InitializeSubchannels{} super.DisplacementChannel{}]
  FeedbackDownInertiaToChannel = Pattern ChannelContribution{
    channel = channel + ["delta"]
    contribution = delta(channelOutput(channel, -1)) * ((0.0 @ 1.0) @ 0.0)}
  FeedbackInertiaToChannel = Pattern ChannelContribution{
    channel = channel + ["delta"]
    contribution = delta(channelOutput(channel, -1))}
  InitializeSubchannels = Pattern [
    ChannelContribution{contribution = channelOutput(channel + ["delta"], 0)}
    SetChannelToReplaceOutput{channel = channel + ["delta"]}
    ChannelContribution{
      channel = channel + ["velocity"]
      contribution = channelOutput(channel + ["delta"], 0) / delta(t)}
    SetChannelToReplaceOutput{channel = channel + ["acceleration"]}
    SetChannelToReplaceOutput{channel = channel + ["velocity"]}
  ]
}
```

C.3.17 VariableSystem Library

```
Library VariableSystem in (POCL AniMoS) {
  InitializeVariables = Pattern Collect(variables{}),
  Pattern signature(variable) variable(Pattern InitializeVariable{}))
  PatternName = Pattern signature(strings, makeName) Concatenate{}
  VariableContext = Pattern signature(variable) VariableOperation(variable, Pattern CustomizedOperations(variable))
}
```

```

VariableDefinition = Pattern signature(variableSystem, desiredVariable) Resolve{
  context = CreateContext{
    parameters = VariableNames{}
    values = VariableDefinitions{}
    pattern = Pattern Parameter(desiredVariable)}
VariableDefinitions = Pattern signature(variableSystem) Resolve{
  pattern = Pattern ExtractEntity(variableSystem{
    VariableSystem = Pattern System{
      children = Pattern TagForExtraction(variables{}))}}
VariableNames = Pattern signature(variableSystem) Collect(
  VariableDefinitions{},
  Pattern signature(variable) variable(Pattern variableName))
VariablesContext = Pattern signature(variables) Combine(Resolve{
  context = CreateContextArray{
    parameters = ["variable"]
    values = [variables{}]}
  pattern = Pattern VariableContext{}})
VariableSystem = Pattern System{
  children = Pattern [
    InitializeVariables{}
    Resolve{
      context = VariablesContext{}
      pattern = outer.children}}}
}

```

C.3.18 GradualActivation Library

```

Library GradualActivation in (POCL AniMoS) {
  DefaultTransition = Pattern SmoothTransition{
    activationTime = activationTime:=(1)
    deactivationTime = deactivationTime:=(activationTime:=(1))}
  GradualActivation = Pattern Resolve{
    Transition = Transition:=(DefaultTransition)
    pattern = Pattern Transition{channel = gradualActivationChannelName}}
  LinearTransition = Pattern [
    ChannelContribution{
      contribution = if(activating, delta(t) / activationTime, (-1 * delta(t)) / deactivationTime)}
    ChannelContribution{contribution = clampChannelOutput(0.0, 1.0)}}
  smoothstep = Pattern signature(input) if(input == 0, 0, if(input == 1, 1, square(input) * (3 - (2 * input))))
  SmoothTransition = Pattern [
    LinearTransition{channel = channel + ["linear"]}
    ChannelContribution{contribution = smoothstep(channelOutput(channel + ["linear"], 0))}
    ChannelContribution{contribution = clampChannelOutput(0.0, 1.0)}
    SetChannelToReplaceOutput{}}
}

```

C.3.19 ActivationOrdering Library

```

Library ActivationOrdering in (POCL AniMoS) {
  ActivationOrdering = Pattern signature(systemNames) Resolve{
    activationChannelName = activationChannelName:=["activated"]
    requestActivationChannelName = requestActivationChannelName:=["activate"]
    pattern = Pattern [InjectCanTriggerCondition{} PropagateCanTriggerConditions{} TriggerConditions{}]}
}

```

```

ConnectCanTriggerCondition = Pattern [
  ChannelContribution{
    channel = targetName + ["canActivate"]
    contribution = channelOutput(sourceName + ["canActivate"], 0)}
  ChannelContribution{
    channel = targetName + ["canActivate"]
    contribution = not(channelOutput(sourceName + activationChannelName, 0))}]
FirstElement = Pattern signature(elements) CopyArray(elements, 1, 1)
InjectCanTriggerCondition = Pattern ChannelContribution{
  channel = FirstElement(systemNames) + ["canActivate"]
  contribution = activation}
PropagateCanTriggerConditions = Pattern Resolve{
  context = CreateContextArray{
    parameters = ["sourceName" "targetName"]
    values = [WithoutLastElement(systemNames)WithoutFirstElement(systemNames)]}
  pattern = Pattern ConnectCanTriggerCondition{}}
TriggerCondition = Pattern [
  SetChannelToAndContributions{channel = systemName + ["canActivate"]}
  SetChannelToAndContributions{channel = systemName + activationChannelName}
  ChannelContribution{
    channel = systemName + activationChannelName
    contribution = channelOutput(systemName + ["canActivate"], 0)}
  ChannelContribution{
    channel = systemName + activationChannelName
    contribution = channelOutput(systemName + requestActivationChannelName, 0)}]
TriggerConditions = Pattern Resolve{
  context = CreateContextArray{
    parameters = ["systemName"]
    values = [systemNames]}
  pattern = Pattern TriggerCondition{}}
WithoutFirstElement = Pattern signature(elements) CopyArray(elements, 2, ArraySize(elements))
WithoutLastElement = Pattern signature(elements) CopyArray(elements, 1, ArraySize(elements) - 1)
}

```

C.3.20 FiniteStateMachine Library

```

Library FiniteStateMachine in (POCL AniMoS) {
  ActiveVariableIdentifier = Pattern stateName + ["enabled"]
  ConditionalTransition = Pattern [
    VariableOperation(Parameter(stateName), Pattern Disable(afterEnabled))
    Resolve{
      condition = channelOutput(stateName + ["condition"])
      pattern = Pattern [
        VariableOperation(Parameter(trueState), Pattern Enable(and([afterEnabled condition])))
        VariableOperation(Parameter(falseState), Pattern Enable(and([afterEnabled not(condition)])))]
      SetChannelToReplaceOutput{channel = stateName + ["condition"]}
      ChannelContribution{
        channel = stateName + ["condition"]
        contribution = condition}}
  EnablingVariableForState = Pattern signature(stateName) System{
    systemName = stateName
    children = Pattern EnablingVariable{enablingScheme = Pattern EnabledUntilDisabled{}}}
  EnablingVariables = Pattern signature(stateName) Resolve{
    context = CreateContextArray{
      parameters = ["stateName"]
      values = [StateNames{}}]
    pattern = Pattern EnablingVariableForState{}}
  EnablingVariablesContext = Pattern CreateContext{
    parameters = StateNames{}}
    values = EnablingVariables{}}
}

```

```

EventDispatchTransition = Pattern [
  events{}
  System{
    systemName = stateName
    children = Pattern ActivationOrdering{
      systemNames = GetNamesFromEventTransitions{}
      requestActivationChannelName = ["condition"]
      activationChannelName = ["activation"]
      activation = whenEnabled}}}
EventTransition = Pattern signature(condition, toState) [System{
  systemName = stateName + toState
  children = Pattern [
    SetChannelToReplaceOutput{channel = ["condition"]}
    ChannelContribution{
      channel = ["condition"]
      contribution = condition}}}
  Resolve{
    activation = channelOutput((stateName + toState) + ["activation"], -1)
    pattern = Pattern VariableOperation(Parameter(stateName), Pattern Disable(activation)))
  Resolve{
    activation = activation * channelOutput((stateName + toState) + ["activation"], -1)
    pattern = Pattern VariableOperation(Parameter(toState), Pattern Enable(activation))}]
FiniteStateMachine = Pattern System{
  systemName = parentSystemName:=({})
  children = Pattern Resolve{
    context = EnablingVariablesContext{}
    pattern = Pattern [
      VariableOperation{
        variable = Parameter(initialState)
        operation = Pattern SetVariable{}
        value = delta(immediateActivation:=(activation(0))) > 0}
      states{disableWhen = disableWhen:=(not(immediateActivation:=(activation(0))))}}}]
GetNamesFromEventTransitions = Pattern ExtractEntities(
  events{EventTransition = Pattern TagForExtraction(toState)})
ImmediateTransition = Pattern signature(toState) [
  VariableOperation(Parameter(stateName), Pattern Disable(afterEnabled))
  VariableOperation(Parameter(toState), Pattern Enable(afterEnabled))]
NoTransition = Pattern EventDispatchTransition{
  events = Pattern [EventTransition{condition = 0 toState = initialState}]}
SequenceEventActivations = Pattern Resolve{
  pattern = Pattern ChannelContribution{
    channel = {target "activation"}
    contribution = 1 - channelOutput([source "activation"], 0)}
  context = CreateContextArray{
    parameters = ["source" "target"]
    values = [RemoveLast(stateNames)RemoveFirst(stateNames)]}]
StateNames = Pattern ExtractEntities(states{State = Pattern TagForExtraction(stateName)})

```

```

TimedEventTransition = Pattern signature(time, toState) [
  EventTransition{condition = channelOutput((stateName + toState) + "timer", 0) == 0}
  System{
    systemName = stateName + toState
    children = Pattern [
      SetMinimumOutputOfChannel{
        channel = "timer"
        minimumOutput = 0}
      ChannelContribution{
        channel = "timer"
        contribution = weigh(time, delta(whenEnabled) == 1)}
      ChannelContribution{
        channel = "timer"
        contribution = weigh(-1 * channelOutput("timer"), 1 - activation)}
      ChannelContribution{
        channel = "timer"
        contribution = negated(delta(t))}]}}}
}

```

State Library

```

Library State in (POCL AniMoS FiniteStateMachine) {
  InitializeStateActivation = Pattern [
    VariableOperation(Parameter(stateName), Pattern InitializeVariable{})
    VariableOperation(Parameter(stateName), Pattern Disable(disableWhen))]
  State = Pattern [
    InitializeStateActivation{}
    System{
      systemName = stateName
      activation = VariableOperation(Parameter(stateName), Pattern enabled(-1))
      immediateActivation = VariableOperation(Parameter(stateName), Pattern enabled(0))
      disableWhen = not(VariableOperation(Parameter(stateName), Pattern enabled(0)))
      children = effects}
    Resolve{
      pattern = transition
      context = Context {
        whenEnabled = activation(0) * VariableOperation(Parameter(stateName), Pattern enabled(0))
        afterEnabled = activation(0) * VariableOperation(Parameter(stateName), Pattern enabled(-1))
        activation = activation(0) * Resolve{
          pattern = Pattern GetVariable{variableName = ActiveVariableIdentifier{}}
          context = MemoryVariable{}}}
    }
}

```

C.3.21 ControllerModel Library

```

Library ControllerModel in (POCL AniMoS) {
  ControllerModel = Pattern Resolve{
    context = ActionInterfaceContext{}
    pattern = controllers}
  currentPostureIs = Pattern signature(postureName)
    channelOutput{channel = ("PostureController" + postureName) + ["enabled"]}
  EnableController = Pattern signature(controllerName) ChannelContribution{
    channel = controllerName + ["enabled"]
    contribution = activation(0) > 0}
  GetControllerVariable = Pattern signature(controllerName, variableName) Resolve{
    context = ActionController
    pattern = Pattern GetControllerVariable{controller = Parameter(controllerName)}}
  RequestPosture = Pattern signature(postureName) ChannelContribution{
    channel = "PostureController" + postureName
    contribution = activation{}}
}

```

```

SetControllerVariable = Pattern signature(controllerName, variableName) Resolve{
    context = ActionController
    pattern = Pattern SetControllerVariable{controller = Parameter(controllerName)}}
}

```

ActionController Library

```

Library ActionController in (POCL AniMoS ControllerModel) {
    ActionController = Pattern VariableSystem{
        inputs = inputs:={}
        outputs = outputs:={}
        systemName = controllerName
        variables = Pattern inputs{} + outputs{}
        children = effects}
    GetControllerVariable = Pattern signature(controller, variableName)
        VariableOperation(VariableDefinition(controller, variableName), Pattern GetVariable{})
    SetControllerVariable = Pattern signature(controller, variableName, value)
        VariableOperation(VariableDefinition(controller, variableName), Pattern SetVariable{})
    VariableDefinition = Pattern signature(controller, desiredVariable) Resolve{
        inputs = {}
        outputs = {}
        VariableDefinition = outer.VariableDefinition
        pattern = Pattern VariableDefinition(controller, desiredVariable)}
}

```

PostureController Library

```

Library PostureController in (POCL AniMoS ControllerModel) {
    EnableControllerContext = Pattern Context {
        EnableController = LocalOperation( Context {}, Pattern super.EnableController{}, ["controllerName"])}
    isPostureNamed = Pattern signature(postureName) channelOutput(postureName + ["enabled"], 0)
    Posture = Pattern {
        State{
            stateName = postureName
            transition = Pattern EventDispatchTransition{
                events = Pattern Resolve{
                    context = TransitionsContext{}
                    pattern = transitions}}}
        SetChannelToOrContributions{channel = postureName}]
    PostureController = Pattern Resolve{
        context = EnableControllerContext{}
        pattern = Pattern VariableSystem{
            systemName = "PostureController"
            variables = Pattern [EnablingVariable(enablingScheme = Pattern EnabledAlways{})]
            children = Pattern FiniteStateMachine{
                activation = Pattern signature(offset) outer.enabled(offset)
                initialState = startingPosture
                states = postures}}}
    Transition = Pattern signature(condition, toPosture) EventTransition(condition, toPosture)
    TransitionsContext = Pattern Combine(
        Resolve{
            context = CreateContextArray{
                parameters = ["postureName"]
                values = [postures{Posture = Pattern postureName}]}
            pattern = Pattern CreateContext{
                parameters = [Concatenate("want" + postureName) Concatenate("is" + postureName)]
                values = [wantPostureNamed(postureName) isPostureNamed(postureName)]})
        wantPostureNamed = Pattern signature(postureName) channelOutput(postureName, 0)
    }
}

```

C.3.22 ActionModel Library

```

Library ActionModel in (POCL AniMoS) {
  actionActivated = Pattern signature(actionName) channelOutput(actionName + ["activation"]) == 1
  ActionBlend = Pattern actions{}
  actionDeactivated = Pattern signature(actionName) channelOutput(actionName + ["activation"]) == 0
  ActionModel = Pattern Resolve{
    actionLayers = actionLayers:={}
    actions = actions:={}
    BlendingScheme = BlendingScheme:=(ImportanceAdjustedChannelContribution)
    context = ControllerInterfaceContext{}
    pattern = Pattern [actions{} actionLayers{}]}
  ActivateAction = Pattern signature(actionName) ChannelContribution{
    channel = actionName + ["active"]
    contribution = activation(-1) > 0}
  ActivationControl = Pattern SmoothActivation{}
  ActivationIntegral = Pattern signature(children) [
    TimeChannel(time = activation * delta(t))
    children{t = channelOutput(actionName + ["time"])}]}
  ActivationTime = Pattern signature(children) [
    TimeChannel(time = delta(t))
    children{t = channelOutput(actionName + ["time"])}]}
  AlwaysActive = Pattern signature(children) [
    SetInitialOutputOfChannel{
      channel = actionName + ["active"]
      initialOutput = 1}
    EdgeActivation{}]}
  ConnectSourceLayerToTargetLayer = Pattern Resolve{
    BlendingScheme = BlendingScheme:=(WeightedChannelContribution)
    context = CreateContextArray{
      parameters = ["dof"]
      values = [dofs]}
    pattern = Pattern Resolve{
      context = BlendingScheme
      pattern = Pattern ChannelOutputWhenInactive{
        channel = targetLayer + dof
        output = channelOutput(sourceLayer + dof, 0)}}}
  ContributeDeltaSourceLayerToTargetLayer = Pattern Resolve{
    BlendingScheme = BlendingScheme:=(NormalizedChannelContribution)
    context = CreateContextArray{
      parameters = ["dof"]
      values = [dofs]}
    pattern = Pattern Resolve{
      context = BlendingScheme
      pattern = Pattern ChannelContribution{
        channel = targetLayer + dof
        contribution = channelOutput(sourceLayer + dof, 0)
        weight = channelOutput(activationChannelIdentifier(sourceLayer + dof), 0)}}}
  ContributeSourceLayerToTargetLayer = Pattern Resolve{
    BlendingScheme = BlendingScheme:=(NormalizedChannelContribution)
    context = CreateContextArray{
      parameters = ["dof"]
      values = [dofs]}
    pattern = Pattern Resolve{
      context = BlendingScheme
      pattern = Pattern ChannelContribution{
        channel = targetLayer + dof
        contribution = channelOutputDeltaTo(channelOutput(sourceLayer + dof, 0))
        weight = channelOutput(activationChannelIdentifier(sourceLayer + dof), 0)}}}
  dofOutput = Pattern signature(channel, offset) channelOutput()

```



```

EdgeActivation = Pattern signature(children) children{
    weight = channelOutput(actionName + ["active"], 0)
    activation = channelOutput(actionName + ["active"], 0)}
ExternalActivation = Pattern signature(children) children{}
ExternalTimeSource = Pattern signature(children) children{}
GetNamesFromActions = Pattern actions{Action = Pattern actionName}
LayerActivationControl = Pattern MultipleActive{}
ResetChannelWhenInactive = Pattern ChannelContribution{
    contribution = weigh(negated(channelOutput{}), activation == 0)}
SmoothActivation = Pattern signature(children) {
    GradualActivation{
        activating = channelOutput(actionName + ["active"], 0)
        gradualActivationChannelName = actionName + ["activation"]}
    children{
        weight = channelOutput(actionName + ["activation"], 0)
        activation = channelOutput(actionName + ["activation"], 0)}}
TimeChannel = Pattern {
    ChannelContribution{
        channel = actionName + ["time"]
        contribution = time * inputScale==(1)}
    ResetChannelWhenInactive{
        channel = actionName + ["time"]
        activation = channelOutput(actionName + ["activation"])}
TimeControl = Pattern ActivationIntegral{}
ZeroTargetLayerDOFDefaults = Pattern Resolve{
    BlendingScheme = BlendingScheme==(WeightedChannelContribution)
    context = createContextArray{
        parameters = ["dof"]
        values = [dofs]}
    pattern = Pattern Resolve{
        context = BlendingScheme
        pattern = Pattern ChannelOutputWhenInactive{
            channel = targetLayer + dof
            output = zero}}}
}

```

Action Library

```

Library Action in (POCL AniMoS ActionModel) {
    Action = Pattern ActivationControl{
        children = Pattern TimeControl{children = effects}}
    EnableController = Pattern outer.EnableController{activation = activation==(weight)}
    RelativeImportance = Pattern children{importance = importance==(1) * scaleFactor}
    SetDOF = Pattern Resolve{
        context = BlendingScheme
        pattern = Pattern ChannelContribution{
            channel = dofName
            contribution = channelOutputDeltaTo(value)}}
    SetDOFDelta = Pattern Resolve{
        context = BlendingScheme
        pattern = Pattern ChannelContribution{
            channel = dofName
            contribution = deltaValue}}
}

```

ActionGroup Library

```

Library ActionGroup in (POCL AniMoS ActionModel) {

```

```

ActionGroup = Pattern [
  actions{BlendingScheme = BlendingScheme::(WeightedChannelContribution)}
  ActivationOrdering{
    systemNames = GetNamesFromActions{}
    requestActivationChannelName = ["active"]
    activationChannelName = ["trigger"]
    activation = true}}
EdgeActivation = Pattern signature(children) children{
  weight = channelOutput(actionName + ["trigger"], 0)
  activation = channelOutput(actionName + ["trigger"], 0)}
SmoothActivation = Pattern signature(children) [
  GradualActivation{
    activating = channelOutput(actionName + ["trigger"], 0)
    gradualActivationChannelName = actionName + ["activation"]}
  children{
    weight = channelOutput(actionName + ["activation"], 0)
    activation = channelOutput(actionName + ["activation"], 0)}}
}

```

ActionLayer Library

```

Library ActionLayer in (POCL AniMoS ActionModel) {
  ActionLayer = Pattern System{
    systemName = layerName
    children = Pattern [LocalDOFs{} LayerActivationControl{} LayerActivation{}]}
  LayerActivation = Pattern [
    SetChannelToOrContributions{channel = ["active"]}
    Resolve{
      context = CreateContextArray{
        parameters = ["actionName"]
        values = [GetNamesFromActions{}]}
      pattern = Pattern ChannelContribution{
        channel = ["active"]
        contribution = channelOutput(actionName + ["active"], 0)}}}
  LocalDOFs = Pattern ExtractEntities(StructuralModel{
    transition = channelOutput(["active"])
    ClampedDOF = Pattern TagForExtraction(outer.ClampedDOF{}))}
  MultipleActive = Pattern actions{}
  OneActive = Pattern ActionGroup{}
}

```

ClampedDOF Library

```

Library ClampedDOF in (POCL AniMoS) {
  ClampedDOF = Pattern signature(dofName, range) [InitializeVariable{variableName = dofName}]
  DefaultCombinationScheme = Pattern NormalizedChannelContribution
  InitializeVariable = Pattern [
    ClampOutputOfChannel{
      channel = variableName
      minimumOutput = ArrayElement(range, 1)
      maximumOutput = ArrayElement(range, 2)}
    Resolve{
      context = ChannelContributionScheme::(DefaultCombinationScheme{})
      pattern = Pattern SetInitialOutputOfChannel{
        channel = activationChannelIdentifier(variableName)
        initialOutput = zero}}}
}
Library ClampedDOF extends super.ClampedDOF in (POCL AniMoS ActionModel ActionLayer) {
  ClampedDOF = Pattern signature(dofName, range) [super.ClampedDOF{}]}

```

```

MaintainActivationDuringTransition = Pattern Resolve{
  context = ChannelContributionScheme==(DefaultCombinationScheme{})
  pattern = Pattern ContributeToActivation{
    channel = activationChannelIdentifier(dofName)
    activation = transition * channelOutput(activationChannelIdentifier(dofName))}}
}

```

C.3.23 BehavioralModel Library

```

Library BehavioralModel in (POCL AniMoS) {
  BehavioralModel = Pattern Resolve{
    behaviors = behaviors==([])
    context = Combine({
      ActionInterfaceContext{}
      ControllerInterfaceContext{}
      ModuleInterfaceContext{}
      SensorFeedbackContext{}
      ModelAttributesContext{}})
    pattern = Pattern {
      InitializeModelAttributes{}
      intentions{}
      behaviors{activation = 1}
      tasks{}
      Resolve{
        pattern = Pattern MakeExternalConnections{modules = tasks}
        context = Module}
      extras{}}}
  BehaviorBlend = Pattern System{systemName = parentSystemName==([])}
  BehaviorValueIdentifier = Pattern [behaviorName "value"]
  InitializeModelAttributes = Pattern Resolve{
    context = attributes
    pattern = Pattern Resolve{
      context = attributeType{}
      pattern = Pattern InitializeVariable{variableName = attributeName}}}
  ModelAttributesContext = Pattern CreateContext{
    parameters =
      Collect(attributes, Pattern signature(attribute) attribute(Pattern NameForGetAttributePattern{})) +
      Collect(attributes, Pattern signature(attribute) attribute(Pattern NameForSetAttributePattern{}))
    values = Collect(attributes,
      Pattern signature(variable) LocalOperation( Context {
        variable = variable,
        Pattern VariableOperation(variable,
          Pattern GetVariable{
            variableName = attributeName}))) + Collect(attributes,
      Pattern signature(variable) LocalOperation( Context {
        variable = variable,
        Pattern VariableOperation(variable,
          Pattern SetVariable{
            variableName = attributeName}))))))
  ModuleInterfaceContext = Pattern Context {
    EnableModule = LocalOperation( Context {}, Pattern super.EnableModule{})
    SetModuleInput = LocalOperation( Context {}, Pattern super.SetModuleInput{})
    GetModuleOutput = LocalOperation( Context {}, Pattern super.GetModuleOutput{})
    GetModuleOutputActivation = LocalOperation( Context {}, Pattern super.GetModuleOutputActivation{})
  NameForGetAttributePattern = Pattern attributeName
  NameForSetAttributePattern = Pattern Concatenate("Set" + attributeName, true)
  SetModuleInput = Pattern signature(moduleName, inputName, value) Resolve{
    context = Module
    pattern = Pattern ToModuleInput{module = Parameter(moduleName)}}
}

```

BehaviorAttribute Library

```

Library BehaviorAttribute extends Variable in (POCL AniMoS BehavioralModel) {
  BehaviorAttribute = Pattern DefaultEntryPattern{}
  GetVariableActivation = Pattern 1.0
  InitializeVariable = Pattern Resolve{
    pattern = Pattern [
      SetInitialOutputOfChannel{initialOutput = initialValue}
      SetMinimumOutputOfChannel{minimumOutput = 0.0}
      ChannelContribution{contribution = growth}
      ChannelContribution{contribution = (channelOutput{} * damping) * -1}]
    context = Context {
      damping = damping:=(zero) * delta(t)
      channel = variableName
      growth = growth:=(zero) * delta(t)
      initialValue = initialValue:=(zero)}}
  SetVariable = Pattern ChannelContribution{
    channel = variableName
    contribution = activation(-1) * (gain:=(1) * delta(t))}
  VariableDefinitionContext = Pattern Combine([
    super.VariableDefinitionContext{variableName = attributeName}
    Context {
      growth = growth
      damping = damping
      attributeName = attributeName
      variableType = Pattern BehaviorAttribute
      attributeType = Pattern BehaviorAttribute}}])
}

```

Behavior Library

```

Library Behavior in (POCL AniMoS BehavioralModel) {
  Behavior = Pattern [
    BehaviorValue{}
    Resolve{
      pattern = BehaviorValueContribution
      context = CreateContextArray{
        parameters = ["contribution"]
        values = {dependencies:={[]}}}
    Resolve{
      parentSystemName = behaviorName
      BehaviorActivationScheme = BehaviorActivationScheme:=(DefaultBehaviorActivationScheme)
      behaviorValue = LocalBehaviorValue{}
      pattern = Pattern BehaviorActivationScheme{}
      context = Context {}}]
  BehaviorActivation = Pattern [
    Resolve{
      triggerChannelIdentifier = [behaviorName "active"]
      activationChannelIdentifier = [behaviorName "activation"]
      context = TriggeredActivation
      pattern = Pattern TriggeredActivation{}}
    ChannelContribution{
      channel = [behaviorName "active"]
      contribution = behaviorValue{} > 0}}
  BehaviorEnabledIdentifier = Pattern [behaviorName "enabled"]
  behaviorValue = Pattern signature(offset) channelOutput(BehaviorValueIdentifier{}, offset:=(0))
  BehaviorValueContribution = Pattern ChannelContribution{channel = UninhibitedBehaviorValueIdentifier{}}
  BehaviorValueIdentifier = Pattern [behaviorName "value"]
  ConditionallySetLevelOfInterest = Pattern LevelOfInterestContribution{
    contribution = if(condition, channelOutputDeltaTo(value), 0)}
}

```

```

ContextForEdgeActivation = Pattern Context {
  bv = LocalOperation( Context {behaviorName = behaviorName}, Pattern behaviorValue{})
  ubv = LocalOperation( Context {behaviorName = behaviorName}, Pattern uninhibitedBehaviorValue{})
  activation = Pattern signature(offset) and([bv() > 0 bv() == ubv()])
ContextForSmoothActivation = Pattern Context {
  previousActivation = channelOutput([behaviorName "activation"], -1)
  activation = channelOutput([behaviorName "activation"], 0)
DefaultBehaviorActivationScheme = Pattern EdgeBehaviorActivation{}
EdgeBehaviorActivation = Pattern Resolve{
  pattern = Pattern Blend{children = effects{}}
  context = ContextForEdgeActivation{}}
LevelOfInterestContribution = Pattern ChannelContribution{channel = LevelOfInterestIdentifier{}}
LevelOfInterestIdentifier = Pattern [behaviorName "levelOfInterest"]
localBehaviorValue = Pattern signature(offset) channelOutput(BehaviorValueIdentifier{ }, offset:=0)
ResetLevelOfInterest = Pattern LevelOfInterestContribution{
  channel = LevelOfInterestIdentifier{ }
  contribution = activation{}}
SmoothBehaviorActivation = Pattern Blend{
  children = {
    BehaviorActivation{ }
    Resolve{
      pattern = Pattern Blend{children = effects{}}
      context = ContextForSmoothActivation{}}}
  uninhibitedBehaviorValue = Pattern signature(offset) channelOutput(UninhibitedBehaviorValueIdentifier{ }, offset:=0)
  UninhibitedBehaviorValueIdentifier = Pattern [behaviorName "uninhibited"]
}

```

BehaviorValue Library

```

Library BehaviorValue in (POCL AniMoS BehavioralModel Behavior) {
  BehaviorValue = Pattern Combine{children = [
    SetChannelToReplaceOutput{channel = BehaviorEnabledIdentifier{}}
    ChannelContribution{
      channel = BehaviorEnabledIdentifier{ }
      contribution = activation(0)}
    SetChannelToReplaceOutput{channel = BehaviorValueIdentifier{}}
    SetMinimumOutputOfChannel{
      channel = BehaviorValueIdentifier{ }
      minimumOutput = 0.0}
    LevelOfInterest{ }
    SetChannelToReplaceOutput{channel = UninhibitedBehaviorValueIdentifier{}}
    SetMinimumOutputOfChannel{
      channel = UninhibitedBehaviorValueIdentifier{ }
      minimumOutput = 0.0}
    ChannelContribution{
      channel = BehaviorValueIdentifier{ }
      contribution = weigh(channelOutput(UninhibitedBehaviorValueIdentifier{ }, 0),
        channelOutput(LevelOfInterestIdentifier{ }, 0) * activation(offset = 0))}]
}

```

```

LevelOfInterest = Pattern {
  ClampOutputOfChannel{
    minimumOutput = 0.0
    channel = LevelOfInterestIdentifier{}
    maximumOutput = 1.0}
  SetInitialOutputOfChannel{
    channel = LevelOfInterestIdentifier{}
    initialOutput = 1}
  ChannelContribution{
    channel = LevelOfInterestIdentifier{}
    contribution = weigh((-1 * decayRate:=0) * delta(t), super.behaviorValue(-1) > 0)}
  ChannelContribution{
    channel = LevelOfInterestIdentifier{}
    contribution = weigh(growthRate:=0) * delta(t), super.behaviorValue(-1) > 0)}
  ChannelContribution{
    channel = LevelOfInterestIdentifier{}
    contribution = weigh(1, and([activation(-1) <= 0 activation(0) > 0]))}
}

```

LocalBehaviorValue Library

```

Library LocalBehaviorValue in (POCL AniMoS BehavioralModel Behavior) {
  LocalBehaviorValue = Pattern PartialResolve{
    context = Context {
      LocalVariableQualifier = LocalVariableQualifier{}
      LocalVariableIdentifier = Pattern LocalVariableQualifier + variableName}
    pattern = Pattern localBehaviorValue{}}
}

```

BehaviorGroup Library

```

Library BehaviorGroup in (POCL AniMoS BehavioralModel) {
  BehaviorGroup = Pattern {
    BehaviorBlend{}
    System{children = Pattern MakeInhibitionConnections{
      behaviorNames = GetNamesFromBehaviors{behaviors = outer.children}}
      systemName = parentSystemName:=({})}}
  GenerateChannelOutputOffsets = Pattern Inject{
    seed = []
    pattern = Pattern signature(result, element) result + element
    array = Resolve{
      pattern = Pattern Repeat{length = pivot array = [0]}
      + Repeat{length = length - pivot array = [-1]}
      context = CreateContextArray{
        values = [[getAttribute(behaviorNames, "size")] Interval(0, getAttribute(behaviorNames, "size") - 1)]
        parameters = ["length" "pivot"]
        fillStyles = ["repeat" "repeat"]}}
  GetNamesFromBehaviors = Pattern ExtractEntities{
    children = behaviors{Behavior = Pattern TagForExtraction(behaviorName)}}
  InhibitBehavior = Pattern ChannelContribution{
    channel = BehaviorValueIdentifier{behaviorName = target}
    contribution = negated(gain) * channelOutput{BehaviorValueIdentifier{behaviorName = source}, offset:=0)}}
}

```

```

MakeInhibitionConnections = Pattern Blend{
  children = Resolve{
    pattern = Pattern InhibitBehavior{}
    context = CreateContextArray{
      values = {behaviorNames behaviorNames gains:={Repeat{
        length = (getAttribute(behaviorNames, "size") - 1) * (getAttribute(behaviorNames, "size") + 1)
        array = [0] + Repeat{
          length = getAttribute(behaviorNames, "size")
          array = [2]} + [0]}
      GenerateChannelOutputOffsets{}}
    parameters = ["source" "target" "gain" "offset"]
    fillStyles = [{"stretch" "repeat" "repeat" "repeat"}]}
}

```

C.3.24 Module Library

```

Library Module in (POCL AniMoS) {
  ConnectModuleOutput = Pattern destinations{
    value = VariableOperation(VariableDefinition(module, outputName), Pattern GetVariable{})
    activation = VariableOperation(VariableDefinition(module, outputName), Pattern GetVariableActivation{}}
  ConnectOutput = Pattern signature(outputName, destinations) ConnectModuleOutput{}
  ConnectToEnable = Pattern signature(sources) sources{
    destinationModule = module
    ToModuleInput = Pattern ToEnableModule{}}
  ConnectToInput = Pattern signature(inputName, sources) sources{destinationModule = module}
  ContextOfActivationVariables = Pattern Context {
    activation = Pattern signature(offset) activate{}}
  ExtractEntityFromModule = Pattern ExtractEntity{children = module{
    Module = Pattern outer.Module{effects = Pattern TagForExtraction{entity = entity{}}}}
  GetModuleOutput = Pattern VariableOperation(VariableDefinition(module, outputName), Pattern GetVariable{})
  GetModuleOutputActivation = Pattern
    VariableOperation(VariableDefinition(module, outputName), Pattern GetVariableActivation{})
  MakeExternalConnections = Pattern Resolve{
    context = CreateContextArray{
      parameters = ["module"]
      values = {modules}}
    pattern = Pattern Resolve{
      context = Context {
        externalConnections = ExtractEntityFromModule{entity = Pattern connections:={[]}}
      }
    pattern = Pattern externalConnections{}}
}

```

```

Module = Pattern VariableSystem{
  systemName = moduleName
  inputs = inputs:={}
  outputs = outputs:={}
  internals = variables:={}
  variables = Pattern ((Resolve{
    context = CreateContextArray{
      parameters = ["variableName"]
      values = [inputs]}
    pattern = Pattern VariableTypeForInput(variableName)} + Resolve{
    context = CreateContextArray{
      parameters = ["variableName"]
      values = [outputs]}
    pattern = Pattern VariableTypeForOutput(variableName)})) + Resolve{
    context = CreateContextArray{
      parameters = ["variableName"]
      values = [internals]}
    pattern = Pattern VariableTypeForInternal(variableName)} + [
    EnablingVariable{}}
  connections = connections:={}
  children = Pattern Resolve{
    context = ContextOfActivationVariables{
      pattern = effects}}
ModuleReset = Pattern effects{
  activation = and([activate{offset = -1} not(activate{offset = 0})])}
Output = Pattern signature(moduleName, outputName) ConnectModuleOutput{
  module = Parameter(moduleName)
  destinations = Pattern ToModuleInput{module = destinationModule}}
ToDisableModule = Pattern VariableOperation(VariableDefinition(module, "enable"), Pattern Disable{})
ToEnable = Pattern signature(moduleName) ToEnableModule{module = Parameter(moduleName)}
ToEnableModule = Pattern VariableOperation(VariableDefinition(module, "enable"), Pattern Enable{})
ToInhibitModuleOutput = Pattern
  VariableOperation(VariableDefinition(module, outputName), Pattern InhibitVariable{})
ToInhibitOutput = Pattern
  signature(moduleName, outputName) ToInhibitModuleOutput{module = Parameter(moduleName)}
ToInput = Pattern signature(moduleName, inputName) ToModuleInput{module = Parameter(moduleName)}
ToModuleInput = Pattern VariableOperation(VariableDefinition(module, inputName), Pattern SetVariable{})
ToSuppressInput = Pattern signature(moduleName, inputName) ToSuppressModuleInput{
  module = Parameter(moduleName)}
ToSuppressModuleInput = Pattern
  VariableOperation(VariableDefinition(module, inputName), Pattern SuppressVariable{})
VariableTypeForInput = Pattern signature(variableName) Resolve{
  pattern = VariableReference(variableName, SuppressibleVariable)}
VariableTypeForInternal = Pattern signature(variableName) Resolve{
  pattern = VariableReference(variableName, MemoryVariable)}
VariableTypeForOutput = Pattern signature(variableName) Resolve{
  pattern = VariableReference(variableName, InhibitVariable)}
}

```


Appendix D

Selected Patterns of the APE Libraries

D.1 APE Library

```
Library APE in (POCL) {  
  AccelerationDueToGravity = Pattern DownDirection{} * 9.8  
  APE = Pattern signature(entity) entity{  
    ArticulatedFigure = Primitive ArticulatedFigure  
    World = Primitive World  
  }
```

D.1.1 ArticulatedObject Library

```
Library ArticulatedObject in (POCL APE) {  
  ArticulatedObject = Pattern ArticulatedFigure{  
    name = objectName  
    connectors = Pattern ViB2VRML{  
      model = Pattern ConnectionModel {}  
    }  
    geometry = Pattern VRML{  
      model = Pattern GeometryModel {}  
    }  
    animation = Pattern AniMoS{  
      model = Pattern AnimationModel {}  
    }  
  }  
  AnimationModel = Pattern [MotionModel{} ActionModel{}]
```

D.1.2 Character Library

```
Library Character extends ArticulatedObject in (POCL APE) {  
}
```

AnimationModel Library

```
Library AnimationModel extends ViBS in (POCL APE Character) {
```

```

AnimationModel = Pattern [
    StructuralModel{}
    StateFeedbackModel{}
    MotionModel{}
    ControllerModel{}
    ActionModel{}
    BehavioralModel{}
    SensorFeedbackModel{}
    CommandModel{}]

StateFeedbackContext = Pattern Context {
    downDirection = channelOutput(["direction" "down"], 0)
    backwardDirection = channelOutput(["direction" "backward"], 0)
    forwardDirection = channelOutput(["direction" "forward"], 0)
    upVelocity = channelOutput(["velocity" "up"], 0)
    leftVelocity = channelOutput(["velocity" "left"], 0)
    forwardVelocity = channelOutput(["velocity" "forward"], 0)
    backwardVelocity = channelOutput(["velocity" "backward"], 0)
    totalVelocity = channelOutput(["location" "velocity"], 0)
    downVelocity = channelOutput(["velocity" "down"], 0)
    rightVelocity = channelOutput(["velocity" "right"], 0)
    rightDirection = channelOutput(["direction" "right"], 0)
    currentLocation = channelOutput(locationChannel)
    upDirection = channelOutput(["direction" "up"], 0)
    leftDirection = channelOutput(["direction" "left"], 0)}

StateFeedbackModel = Pattern [
    CreateReferenceDirectionChannels{}
    CreateReferenceVelocityChannels{}]
}

```

BehavioralModel Library

Library BehavioralModel extends super.BehavioralModel in (POCL APE Character AnimationModel) {

```

ApproachObject = Pattern Module{
  moduleName = "ApproachObject"
  EnablingScheme = Pattern EnabledWhileActivated{}
  inputs = ["object" "type"]
  outputs = ["target" "targetType" "approaching" "targetLocation"]
  approaching = ActivationVariable
  connections = Pattern [
    ConnectOutput("approaching", Pattern ToEnable("ApproachLocation"))
    ConnectOutput("targetLocation", Pattern ToInput("ApproachLocation", "desiredLocation"))
    ConnectOutput("approaching", Pattern ToEnable("LookAtObject"))
    ConnectOutput("target", Pattern ToInput("LookAtObject", "object"))
  ]
  effects = Pattern [
    FiniteStateMachine{
      initialState = "Start"
      states = Pattern [
        State{
          stateName = "Start"
          effects = Pattern []
          transition = Pattern EventDispatchTransition{
            events = Pattern [EventTransition(not(object{} == zero), "Focus")]
          }
        }
        State{
          stateName = "Focus"
          effects = Pattern [
            SetTarget{value = object{}}
            SetTargetType{value = type{}}
            SetTargetLocation{value = sensedObjectEntryPoint(object{})}
            SetApproaching{}
          ]
          transition = Pattern ImmediateTransition("Approach")
        }
        State{
          stateName = "Approach"
          effects = Pattern SetApproaching{}
          transition = Pattern EventDispatchTransition(events = Pattern [
            EventTransition(or([not(object{} == target{}) not(type{} == targetType{})]), "Start")
          ])
        }
      ]
    }
  ]
}

```

ApproachObjectOverride Library

```

Library ApproachObjectOverride in (POCL APE Character AnimationModel BehavioralModel) {
  ActiveState = Pattern State{
    stateName = stateName:="Active"
    effects = ActiveEffects:=(Pattern DefaultActiveEffects{})
    transition = Pattern EventDispatchTransition{
      events = Pattern [
        EventTransition{
          condition = or([objectChanged{} typeChanged{}])
          toState = toState:="Record"
        }
      ]
    }
  }
  ApproachObjectOverride = Pattern Module{
    EnablingScheme = Pattern EnabledWhileActivated{}
    inputs = inputs{}
    outputs = outputs{}
    active = ActivationVariable
    arrived = ActivationVariable
    closingIn = ActivationVariable
    effects = Pattern DefaultEffects{}
  }
}

```

```

ApproachState = Pattern State{
  stateName = "Approach"
  effects = Pattern [SetActive{}]
  transition = Pattern EventDispatchTransition{
    events = Pattern [
      EventTransition{
        condition = not(target{} == object{})
        toState = initialState
      }
      EventTransition{
        condition = or({atSubtarget{} closingIn{}})
        toState = "Done" }}}}
atSubtarget = Pattern equalPointsOnPlane{
  normal = upDirection{}
  point1 = currentLocation{}
  point2 = externalChannelOutput(subtarget{offset = 0}, ["location"])
  equalityThreshold = 0.05}
connections = Pattern [
  ConnectToEnable(Pattern Output("ApproachObject", "approaching"))
  ConnectToInput("type", Pattern Output("ApproachObject", "targetType"))
  ConnectToInput("object", Pattern Output("ApproachObject", "target"))
  ConnectOutput("subtarget", Pattern ToSuppressInput("ApproachObjectEntryPoint", "object"))
  ConnectOutput("active", Pattern ToInhibitOutput("ApproachObject", "approachTarget"))
  ConnectToInput("closingIn", Pattern Output("ApproachLocation", "closeIn"))]
DefaultActiveEffects = Pattern FiniteStateMachine{
  initialState = "RecordSubtarget"
  states = Pattern [ RecordSubtargetState{} ApproachState{}. DoneState{}}]
DefaultEffects = Pattern FiniteStateMachine{
  initialState = "Record"
  states = Pattern [RecordState{} TestState{} ActiveState{} WaitState{}}]
DoneState = Pattern State{
  stateName = "Done"
  effects = Pattern [SetActive{}]
  transition = Pattern EventDispatchTransition{
    events = Pattern [
      EventTransition{
        condition = 0
        toState = "RecordTargetLocation" }}}}
inputs = Pattern ["object" "type" "closingIn"]
objectChanged = Pattern not(object{offset = 0} == object{})
outputs = Pattern ["active" "target" "subtarget" "arrived"]
RecordState = Pattern State{
  stateName = stateName:="Record")
  effects = Pattern [SetTarget{value = object{}}]
  transition = Pattern ImmediateTransition{toState = toState:="Test")}}
RecordSubtargetState = Pattern State{
  stateName = "RecordSubtarget"
  effects = Pattern [SetActive{} SetSubtarget{value = desiredSubtarget{}}]
  transition = Pattern ImmediateTransition{toState = "Approach")}}
RecordTargetLocationState = Pattern State{
  stateName = "RecordTargetLocation"
  effects = Pattern [SetActive{} SetTargetLocation{value = desiredTargetLocation{}}]
  transition = Pattern ImmediateTransition{toState = "Approach")}}
RecordTargetState = Pattern State{
  stateName = stateName:="RecordTarget")
  effects = Pattern [SetTarget{value = object{}}]
  transition = Pattern ImmediateTransition{toState = toState:="Test")}}

```

```

TestState = Pattern State{
  stateName = stateName:=( "Test" )
  effects = Pattern []
  transition = Pattern EventDispatchTransition{
    events = Pattern [
      EventTransition{
        condition = TestStateCondition{}
        toState = activeState:=( "Active" )}}
      EventTransition{
        condition = not(TestStateCondition{})
        toState = inactiveState:=( "Wait" )}}}]
  TestStateCondition = Pattern and({type{} == desiredObjectType{} externalChannelOutput(
    object{}, {"objectID"}) == desiredObjectID{}})
  typeChanged = Pattern not(type{offset = 0} == type{})
  WaitState = Pattern State{
    stateName = stateName:=( "Wait" )
    effects = Pattern []
    transition = Pattern EventDispatchTransition{
      events = Pattern [
        EventTransition{
          condition = or({objectChanged{} typeChanged{}})
          toState = toState:=( "Record" )}}}]
}

```

TargetingController Library

```

Library TargetingController in (POCL APE Character AnimationModel ControllerModel) {
  Dimensions = Pattern Resolve{
    context = CreateContextArray{
      parameters = ["dimensionName" "dimensionNormal" "referenceLocation" "referenceDirection"]
      values = [dimensions normals referenceLocations:=[referenceLocation] referenceDirections:=[referenceDirection]]}
    pattern = Pattern Resolve{
      context = Context {}
      pattern = Parameter(Concatenate(dimensionName + "Dimension", true), TargetingDimension)}}
  effects = Pattern [Dimensions{}]
  inputs = Pattern [
    MemoryVariable{variableName = "targetLocation"}
    EnablingVariable{enablingScheme = Pattern EnabledWhileActivated{}}]
  outputs = Pattern []
  TargetingController = Pattern ActionController{}
  TargetingDimension = Pattern System{
    systemName = dimensionName
    children = Pattern [
      SetChannelToReplaceOutput{channel = ["direction"]}
      ChannelContribution{
        channel = ["direction"]
        contribution = normalize((targetLocation(0) - referenceLocation{}) - (dimensionNormal{
          * dot(targetLocation(0) - referenceLocation{}, dimensionNormal{}}))}
      SetChannelToReplaceOutput{channel = ["angle"]}
      ChannelContribution{
        channel = ["angle"]
        contribution = angleBetweenDirections(channelOutput("direction", 0),
          referenceDirection{}) * if(distanceSquaredBetween(dimensionNormal{
            cross(channelOutput("direction", 0), referenceDirection{}}) < 1, -1, 1)}}]
}

```

HeadTargetingController Library

```

Library HeadTargetingController extends TargetingController in (POCL APE Character AnimationModel ControllerModel) {

```

```

effects = Pattern super.effects{} + [
  SetChannelToReplaceOutput{channel = ["leftTurnAngle"]}
  SetLeftTurnAngle{value = channelOutput(["horizontal" "angle"], 0)}]
HeadTargetingController = Pattern TargetingController{
  controllerName = "HeadTargetingController"
  dimensions = ["horizontal" "vertical"]
  normals = {upDirection{}leftDirection{}}
  referenceLocation = currentLocation{}
  referenceDirection = forwardDirection{}}
outputs = Pattern super.outputs{} + [ChannelVariable{variableName = "leftTurnAngle"}]
}

```

CycleController Library

```

Library CycleController in (POCL APE Character AnimationModel ControllerModel) {
  CycleController = Pattern ActionController{
    controllerName = controllerName:="("CycleController")"
    effects = Pattern [
      SetChannelToReplaceOutput{channel = "phase"}
      SetPhase{
        value = weigh(i mod(phase() + ((delta(t) * tempo(0)) * (2 * pi{})), 2 * pi{}), enabled{}})
      SetTempoDefault(defaultTempo:=(1))
    ]
    inputs = Pattern [
      EnablingVariable{enablingScheme = Pattern EnabledWhileActivated{}}
      BlendedVariable{variableName = "tempo"}]
    outputs = Pattern [ChannelVariable{variableName = "phase"}]
  }
}

```

ScanningController Library

```

Library ScanningController extends CycleController in (POCL APE Character AnimationModel ControllerModel) {
  effects = Pattern super.effects{enabled = and([enabled{} not(reset{}))]} + [
    SetScanning{
      value = not(scanning{})
      activation = delta(phase{}) < 0}
    SetScanning{
      value = 1
      activation = and([not(enabled(-1)) enabled(0)])}
    SetScanning{
      value = 1
      activation = and([reset() enabled(0)])}
  ]
  inputs = Pattern super.inputs{} + [
    MemoryVariable{variableName = "wide"}
    ActivationVariable{variableName = "reset"}]
  outputs = Pattern super.outputs{} + [MemoryVariable{
    variableName = "scanning"
    initialValue = 1}]
  ScanningController = Pattern CycleController{
    controllerName = "ScanningController"
    defaultTempo = 0.5}
}

```

D.1.3 Robby Library

```

Library Robby extends Character in (POCL APE) {

```

```

Robby = Pattern Creature{
  creatureName = "Robby"
  sensedObjects = ["foodSource" "restArea" "sleepArea" "entertainment"]
  contactPoints = [{"Left" "Foot" "location"} {"Right" "Foot" "location"}]
  constraints = Pattern [SurfaceContact SurfaceFriction SlideToLocation ForwardCollision]
  controllers = Pattern [PostureModel ScanningController BodyCycleController HeadTargetingController
    SittingController BodyTargetingController RightHandTargetingController]
  actions = Pattern [PostureLayer LocomotionLayer LookLayer ReachWithRightHand]
  behaviors = Pattern [RememberObject EngageObject ApproachObject ApproachObjectEntryPoint
    ApproachFridge ApproachTV OperateObject AccessFoodSource UseEntertainment
    FindObject ScanArea Posture LookAtObject Move MoveForward
    MoveToLocation StandUp Falling FaceLocation ApproachLocation EnterSleepArea
    SitDown SitOnObject GainSleep LoseRest]
  intentions = Pattern [IntentionsModel]
  attributes = Pattern [
    BehaviorAttribute("fatigued", 0.2, 0)
    BehaviorAttribute("hungry", 0.4, 0)
    BehaviorAttribute("sleepy", 0.1, 0)]
  LocomotionLayer = Pattern [
    ActionLayer{
      layerName = "Locomotion"
      actions = [AbsorbFall Stopped Walk TurnToLocation StepToLocation]
      LayerActivationControl = Pattern OneActive{}}
    ZeroTargetLayerDOFDefaults{
      targetLayer = "Locomotion"
      dofs = StructuralDOFs{} + [{"orientation" "yaw"}]}
    ContributeSourceLayerToTargetLayer{
      sourceLayer = "Locomotion"
      targetLayer = []
      dofs = StructuralDOFs{}}
    ContributeDeltaSourceLayerToTargetLayer{
      sourceLayer = "Locomotion"
      targetLayer = []
      dofs = [{"orientation" "yaw"}]}
    SetChannelToReplaceOutput{
      channel = [{"Locomotion"} + orientationChannel] + [{"yaw"}]}
    TurnToLocation = Pattern Action{
      actionName = "TurnToLocation"
      activationTime = 0.5
      kneeRange = kneeRange:=(0.35)
      turnRate = turnRate:=(2)
      deactivationTime = 0.1
      maxTurnRate = maxTurnRate:=(2)
      phase = GetControllerVariable("BodyCycleController", "phase")
      TimeControl = Pattern ExternalTimeSource{}
      effects = Pattern [
        SetDOFDelta{
          dofName = orientationChannel + [{"yaw"]}
          deltaValue = maximum(
            minimum((GetControllerVariable("BodyTargetingController", "leftTurnAngle") / delta(t)) * 0.9,
              maxTurnRate{}), -1 * maxTurnRate{})
        }
        EnableController{controllerName = "BodyFocusController"}
        EnableController{controllerName = "BodyCycleController"}
        SetDOF{
          dofName = ["Left" "Knee" "flexion"]
          value = maximum(cos(phase) * kneeRange, 0)}
        SetDOF{
          dofName = ["Right" "Knee" "flexion"]
          value = maximum((cos(phase) * -1) * kneeRange, 0)}}]

```

```

PostureModel = Pattern PostureController{
  startingPosture = "Standing"
  postures = Pattern [
    Posture{postureName = "Standing"
      effects = Pattern [
        ActivateAction(["PostureActions" "Stand"])
        EnableController("SurfaceContact")
        EnableController("SurfaceFriction")
      ]
      transitions = Pattern [
        Transition(wantMoving{}, "Moving")
        Transition(wantSitting{}, "Sitting")
        Transition(wantFalling{}, "Falling")
        Transition(wantClosingIn{}, "ClosingIn")
      ]
    }
    Posture{postureName = "Falling"
      effects = Pattern [
        ActivateAction(["PostureActions" "Fall"])
        EnableController("SurfaceContact")
      ]
      transitions = Pattern [
        Transition(not(wantFalling{}), "Standing")
      ]
    }
    Posture{postureName = "Moving"
      effects = Pattern [
        ActivateAction(["PostureActions" "Stand"])
        EnableController("SurfaceContact")
        EnableController("SurfaceFriction")
        EnableController("ForwardCollision")
      ]
      transitions = Pattern [
        Transition(not(wantMoving{}), "Standing")
        Transition(wantFalling{}, "Falling")
        Transition(wantClosingIn{}, "ClosingIn")
      ]
    }
    Posture{postureName = "ClosingIn"
      effects = Pattern [
        ActivateAction(["PostureActions" "Stand"])
        EnableController("SurfaceContact")
        EnableController("SlideToLocation")
        EnableController("ForwardCollision")
      ]
      transitions = Pattern [
        Transition(not(wantClosingIn{}), "Standing")
        Transition(wantFalling{}, "Falling")
      ]
    }
    Posture{postureName = "Sitting"
      effects = Pattern [
        ActivateAction(["PostureActions" "Sit"])
        EnableController("SittingController")
        ActivateAction(actionName = ["Locomotion" "Stopped"])
      ]
      transitions = Pattern [
        Transition(wantStanding{}, "GettingUp")
        Transition(wantFalling{}, "Falling")
      ]
    }
    Posture{postureName = "GettingUp"
      effects = Pattern [
        ActivateAction(["PostureActions" "Stand"])
        EnableController("SittingController")
      ]
      transitions = Pattern [
        Transition(actionDeactivated(["PostureActions" "Sit"], "Standing"))
      ]
    }
  ]
}

```

ApproachTV Library

Library ApproachTV extends ApproachObjectOverride in (POCL APE Robby AnimationModel BehavioralModel) {


```

ApproachTV = Pattern ApproachObjectOverride{
  operate = ActivationVariable
  moduleName = "ApproachTV"
  concernType = ObjectTypes(Pattern concernID{})
  desiredObjectType = ObjectTypes(Pattern entertainmentID)
  desiredObjectID = ObjectIDs(Pattern tvObjectID{})
  desiredSubtarget = Pattern externalChannelOutput(target{}, ["Knob"])
  connections = Pattern super.connections{} + [
    ConnectOutput("operate", Pattern ToEnable("OperateObject"))
    ConnectOutput("target", Pattern ToInput("OperateObject", "object"))
    ConnectOutput("switchLocation", Pattern ToInput("OperateObject", "switchLocation"))
    ConnectOutput("turnOn", Pattern ToInput("OperateObject", "turnOn"))
  ]
  DoneState = Pattern State{
    stateName = "Done"
    effects = Pattern [
      SetActive{}
      SetOperate{}
      SetSwitchLocation{value = externalChannelOutput(target{}, ["Knob" "location"])}
      SetTarget{value = object{}}
      SetTurnOn{value = type{} == desiredObjectType{}}
    ]
    transition = Pattern NoTransition{}
    outputs = Pattern super.outputs{} + ["operate" "switchLocation" "turnOn" "target"]
    TestStateCondition = Pattern or([
      and([super.TestStateCondition{
        not(externalChannelOutput(object{}, ["activated"])))
      }
      and([super.TestStateCondition{desiredObjectType = concernType{}}
        externalChannelOutput(object{}, ["activated"])]))
    ])
  }
}

```