ARCHITECTING RUBE WORLDS:
A METHODOLOGY FOR CREATING VIRTUAL ANALOG DEVICES AS METAPHORICAL REPRESENTATIONS OF FORMAL SYSTEMS

By

KRISTIAN LINN DAMKJER

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2003

Dedicated to my wife Bryony,
my parents Keith and Deborah Damkjer,
and my sister April Derato,
for their constant support and encouragement

ACKNOWLEDGMENTS

I would like to sincerely thank my advisor and committee chairman, Dr. Paul A. Fishwick, for his constant instruction, guidance, and support throughout my graduate studies at the University of Florida. I would also like to extend my gratitude to Dr. Abdelsalam Ali Helal and Dr. Joachim Hammer for their time and interest in serving on my thesis committee.

I sincerely appreciate the time and devotion of my companions on the University of Florida *RUBE* Project, Mr. Minho Park, Mr. Jinho Lee, and Ms. Hyunju Shim, who made this research truly an enjoyable experience.

Many thanks go to my fellow graduate students in the digital arts and sciences program, especially Mr. John Hays and Mrs. Joella Walz, for their companionship and support through the highs and lows of graduate studies. I will never forget their friendship.

I am deeply grateful to my parents, Keith and Deborah Damkjer, for their constant prayer, support and encouragement, and to my wife, Bryony, whose love, support, and willingness to share these last two wonderful, though often trying, years with me will be a treasured memory for years to come.

Finally, I owe an extreme debt of gratitude to everyone who has prayed for me throughout my academic endeavors and to God for His many blessings; may He continue to guide me and my family throughout our lives.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

ARCHITECTING RUBE WORLDS:
A METHODOLOGY FOR CREATING VIRTUAL ANALOG DEVICES AS
METAPHORICAL REPRESENTATIONS OF FORMAL SYSTEMS

By

Kristian Linn Damkjer

May, 2003

Chair: Paul A. Fishwick
Major Department: Computer and Information Science and Engineering

Several modeling and simulation toolkits have been created to help facilitate the model development process and aid in extending models and simulations into new arenas. New ways of visualizing models and simulations have likewise been developed to take advantage of technologies available in these new arenas. *RUBE* is a project that uses a combination of XML technologies, Java, JavaScript, and the Virtual Reality Modeling Language (VRML), to provide a framework for developing dynamic models that run as virtual environments on the Internet. In *RUBE*, model structure is independent of its presentation, which allows for improved sharing of models and also for presentation customization.

My research attempts to establish a methodology for creating presentation worlds that ensures proper correlations between the model content and presentation. My approach uses virtual analog devices and architectural constructs as metaphorical representations of model structure, though I also provide a generalized methodology for

using metaphorical mappings and grouping structures to ensure strong correlations. This research has also resulted in the restructuring of the Multi-model Exchange Language (MXL)—one of several XML applications employed in *RUBE*—for better model content definition, as well as extended support for multi-modeling in *RUBE*.

CHAPTER 1
INTRODUCTION

A Brief History of *RUBE*

The current modeling and simulation endeavor entitled *RUBE* originates from prior research and development that started in the early 1990s with the creation of code libraries—consisting of C source, libraries and executables—entitled SimPack (1990). Significant portions of SimPack were dedicated to handling future event lists and queuing (both of which are bases for discrete event simulation, a popular method for simulating formally defined dynamic systems) [10]. These portions of SimPack underwent several revisions resulting in OOSIM (1995), coded in C++ using the object-oriented paradigm [10], and SimPack J/S (2002), a JavaScript and Java port developed with the intent of supporting web-based simulation and which now serves as the core simulation engine for *RUBE* [16].

As the names suggest, the focus of SimPack and its successors is model simulation, not model development. To augment the simulation toolkits, specifically SimPack and OOSIM, with multi-model development capabilities, the Object-Oriented Physical Multi-modeling (OOPM) toolkit (1996) was developed [9]. This toolkit provided a graphical interface for developing multi-models with support for several model types and is regarded as the predecessor to the *RUBE* Project [14].

The *RUBE* Project began in 2000 as a continuation of research in multi-modeling and web-based modeling and simulation with the initial purpose of facilitating dynamic multi-model construction and component reuse in three-dimensional distributed

immersive environments. To this end the Virtual Reality Modeling Language (VRML) was employed as the primary language for specifying scene geometry and presentation dynamics. Initial work in this arena was done using the VRML "Prototype" extensibility mechanism [10]; however, that approach has recently been replaced with the Extensible Markup Language (XML) based framework that now serves as the basis for *RUBE* [14]. This approach more strongly decouples the model content from the model presentation which in turn allows for better component reuse and customization of model representation.

The initial framework designed by Kim has been modified slightly to accommodate recent developments in our research and the current structure is outlined in Figure 1-1.

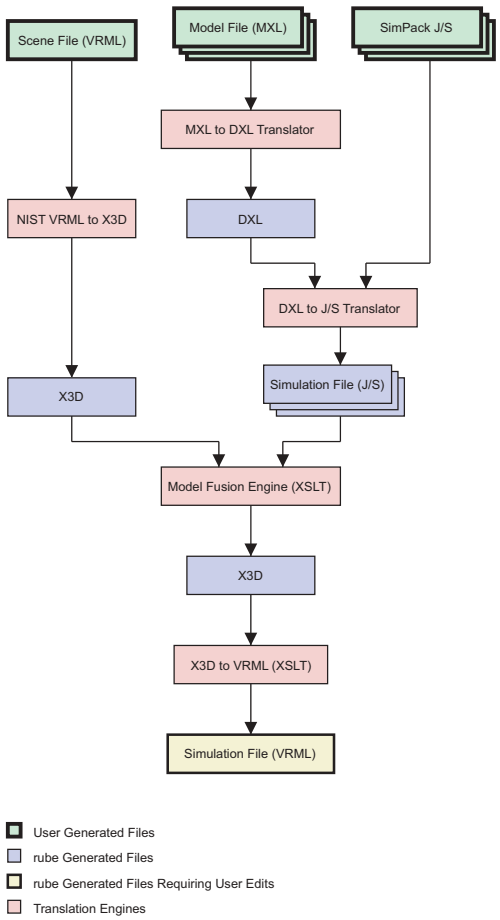

Figure 1-1. *RUBE* architecture

In the *RUBE* pipeline, the developer provides as input scene files describing how the model should be presented, model files defining the model topology, and semantic files defining how the model should execute (simulation rules). These files are then merged using XML Style-sheet Language Transformation (XSLT), Document Object Model (DOM), and custom-developed translation mechanisms to produce a single presentation file with an interface for initializing simulation execution and hooks for simulation output. The developer modifies this file to use the simulation output data to drive the presentation dynamics while the simulation executes.

Challenges

With more robust modeling and simulation toolkits—affording greater possibilities in model definition, presentation, simulation, and distribution—and the fact that model definition is ideally decoupled from its presentation, the need for a formalized methodology for developing models and their corresponding presentations increases.

To this end the primary objectives of this research are to refine the Multi-model Exchange Language (MXL) to remove ambiguities in model definitions and to facilitate language extensions in the form of adding new model types; to extend the *RUBE* framework to allow for model definition and semantic component reuse; to explore model definition development to take advantage of the multi-model paradigm; to explore metaphorical mappings for model representations; and to establish a methodology for creating presentation worlds for models developed with the *RUBE* framework. Auxiliary objectives of this research are to explore the possibility of partially automating the established methodology and to investigate extending the methodology to presentation vehicles other than VRML.

Organization of Thesis

A philosophical background of the *RUBE* approach to modeling and simulation is discussed in Chapter 2. Chapter 3 details the revisions made to MXL and the model semantics definition structure. Chapter 4 details the extensions to the *RUBE* framework to fully support multi-modeling. Chapter 5 explores model definition strategies. Chapter 6 outlines metaphorical mappings to use in model representations. Chapter 7 discusses the methodology to be used in presentation development. Chapters 8 and 9 are summaries of case-study implementations. Conclusions, future work, and auxiliary objectives are discussed in Chapter 10.

CHAPTER 2
BACKGROUND

Mathematics as a Language

Mathematics is often thought of as an eloquent language, highly formalized and structured, able to capture endless representations of phenomena in the world around us succinctly and efficiently. It has its own syntax, grammar, and an alphabet that is the product of centuries of abstraction and stylization due to economics and efficiency [17]. As higher mathematics has evolved, new symbols, syntax, and grammar have been created to extend the mathematical language, albeit non-uniformly and non-universally, resulting in "dialects" of mathematics. In general, however, mathematics is still regarded as a fairly universal language, allowing for the exchange of formalized ideas and concepts in spite of socio-economic as well as natural language barriers.

The power of mathematics to represent formal structures is inherent in its abstract nature. However, with this power of generalization comes a loss of inherent comprehension. Gennady Uzilevsky's hierarchy of languages, which organizes languages and codes based on their inherent comprehensibility among humans, places mathematics into categories that are among the least inherently comprehensible: iconic languages, symbols, and meta-languages. Hughes observes the following when commenting on this hierarchy:

> Only the top five languages are what we think of as "languages": these are the least powerful of the 12 . . . . Most human communication is done by the lower seven, of which the most easily named (in decreasing order of power) are gesture, images, movements, emotional conditions, music and color. [13:180]

Thus, mathematics should be augmented with those languages that are more universally comprehensible, a feat we accomplish or observe readily in literature, oration, and fine art.

Mathematics in Model Design and Simulation

> To *model* is to abstract from reality a description of a dynamic system. Modeling serves as a language for describing systems at some level of abstraction or, additionally, at multiple levels of abstraction. . . . Models are used for the purpose of communicating with each other—the alternative being that two people wishing to discuss dynamics would be forced to work with the real system under investigation. [8:27]

Mathematics, because of its powerful ability to generalize formal structures, is a natural choice for the language of modeling. It is very simple, in mathematical terms, to describe well defined dynamic systems as sets of quantitative, stochastic, and logical relations: The resulting deterministic system is defined as the following 7-tuple [8:46-47]:

$\langle T, U, Y, Q, \Omega, \delta, \lambda \rangle$

- T : the *time* set. For continuous systems $T = \mathbb{R}$ (the real numbers), and for discrete systems $T = \mathbb{Z}$ (the integers).
- U : the *input* set. Contains the possible values of the input to the system.
- Y : the *output* set.
- Q : the *state* set.
- Ω : the set of *admissible* (or *acceptable*) *input* functions. $T \to U$
- δ : the *transition* function. $\delta : Q \times T \times T \times U \to Q$
- λ : the *output* function. $\lambda : Q \to Y$

Through mathematics we are able to not only formalize the structure of dynamic systems, but also to virtually execute and analyze the systems. We can easily and safely

manipulate aspects of the system to observe how the system responds and behaves as a whole. We can apply these observations back to the real-world phenomena being modeled, provided that the model is accurate.

Despite this power, the primary purpose for modeling is not simply for description and analysis of a system, but for communication with others. Unfortunately, the traditional way of representing mathematical concepts with calligraphic notation often fails in this endeavor [11]. We should attempt then to extend our representation of mathematical models. After all, the exchange of ideas is the *raison d'être* of modeling. This extension of the mathematical model is the basis for the University of Florida's *RUBE* Project [5, 8, 12]. Mathematics, in the traditional sense, has been used to represent both the content and presentation of model information; *RUBE* [5, 12] attempts to separate one from the other and allow users to interpret models using their own presentation schemas [10].

Schemas

RUBE [5, 12] is perhaps unique in its approach to modeling and simulation in that it takes a narrative approach to representing models. By doing so it attempts to draw on schemas not only to improve comprehension of models, but also to create a synergistic representation that addresses both of Bruner's "two modes of thought".

> There are two modes of cognitive functioning . . . each producing distinctive ways of ordering experience, of constructing reality. The two (though complementary) are irreducible to one another. Efforts to reduce one mode to the other or to ignore one at the expense of the other inevitably fail to capture the rich diversity of thought. [2:11]

The two modes of cognitive functioning are the *paradigmatic*, which attempts to achieve an ideal formalization or mathematical explanation through categorization and conceptualization of a system, and the *narrative*, which instead deals with the "human or

human-like intention and action and the vicissitudes and consequences that mark their course." [2:13]

By incorporating schema and narrative in our model representation we are attempting to improve comprehension of a model by providing an immediately recognizable and familiar framework for the user. To *successfully* incorporate schemas, however, we must first make some observations about how they are defined and operate, then establish a means of merging the narrative structures with the paradigmatic ones.

There are several parallels between Bruner's schema definition and our formalized model definition. Schemas are composed of states, crises, and redresses with the possibility for cycles and branches in the narrative structure. The result of realizing this structure is story [2]. Similarly, recall that dynamic systems are composed of states, transitions, and next-states, and the result of simulating the system is output. The narrative steady state is breached by some external force thus instigating the steady-state–crisis–redress cycle [2]. In much the same way, systems transfer between states based on input. Finally, *both* establish a chronological framework (even if narrative is not presented in a strictly linear fashion). The mapping between the two is fairly direct: schema-states map to system-states, crises map to transitions, redresses map to next-states, steady-state breach forces map to system input, and story maps to system output.

Note also that we can extend this kind of schema mapping to system multi-modeling in which we can have any combination of *multiplicity*, *heterogeneity*, and *hierarchy* of models. *Multiplicity* implies several model *instances*, *heterogeneity* implies several model *types*, and *hierarchy* implies several model *layers* in a nested structure [10]. Black and Bower's observations on *Episodes as Chunks in Narrative Memory*

conform nicely to the multi-model structure. They observe that stories consist of narrative elements that are interconnected to produce a coherent whole. They also observe that the resulting network of interconnected elements can be represented in one of two ways, either as consisting of several clustered sub-groups, or a resulting "macro"-structure and "micro"-structure [1]. Both views parallel the multi-model paradigm and I would further argue that neither is explicitly the normative structure.

To combine the paradigmatic and narrative modes of model representation in *RUBE* we employ yet another narrative device: *metaphor*. We could have similarly chosen to use the mathematical device of *function*. Both provide a direct mapping from one domain to another; however, metaphor is explicitly a one-to-one mapping, a trait that ensures consistency between the narrative schema and the formal system definition. Note that this merge will result in not only a representation of the model output, but also of the model structure and functionality. This representation goes far beyond that of the simple graph or diagrammatic presentation [11]. Once the narrative world and the paradigmatic model have been merged we can proceed to simulate the model. Assuming that the model requires no direct input from the user to execute, what the user will observe are elements of the narrative domain interacting with each other as a representation of the model execution. Users will likely unconsciously compose a story based on the narrative schema to describe and explain the interactions that they observe. This is a result of *causation* and *perceived intention* as well as drawing on relevant scripts within the schema to shape our perception, navigation, and interaction within a given scenario [6, 13].

Causation and Animism

Bruner recounts a discourse by Baron Michotte in which he demonstrates that when subjects observed objects moving with respect to each other within highly limited confines, the subjects impose causality. They saw the objects' movements as affecting and being caused by the movements of other objects in the scene. It was found that the spatial-temporal relationships of the objects could be manipulated to invoke various types of behavior interpretation such as "launching," "dragging," and "deflecting" [2]. To visualize this behavior, consider the classic arcade games of *Pong* and *Arkanoid* in which a ball is "launched" across the screen at a target and is repeatedly "deflected" by other objects in the scene.

Bruner further relates the work of Heider and Simmel who used methods similar to Michotte's to demonstrate the irresistibility of "perceived intention". Subjects again observed a short animated film and again bestowed behavior to the objects, yet this time the behaviors were perceived as being *intention-driven*. Stewart demonstrated that again, by manipulating the spatial-temporal relationships of the objects, other apparent intention-driven behaviors could be invoked, for example: "searching," "pursuit," and "persistence" [2].

Why are these results so interesting to *RUBE* model applications? The answer is simple, they imply that users will create their own narrative to explain the model presentation in terms that they already understand and will thus be able to achieve an initial *understanding* of the model concept or perhaps a deeper *comprehension*. This understanding will be made further concrete if the user already has a deep understanding of the schema in which the model is being presented.

Recall, however, that one of the greatest strengths of the mathematical model is that it allows us to safely and efficiently manipulate aspects of the system to observe how the system as a whole responds [8]. To achieve this we must have some form of interaction with the schema world and this is achieved through manipulative objects.

Manipulative Objects: Play and the Emotional Factor

By extending our representation of models to include schemas, we have extended the ways in which users can understand our model. By introducing manipulative objects in our schemas, we can allow the user to interact with and affect model execution. The use of manipulative objects as a learning aid has been shown to be a positive influence on learning and comprehension of mathematical concepts in general. This benefit has been shown to hold across grade level, ability level, and topic. The use of manipulative objects alone, however, does not guarantee successful comprehension of concepts [4]. In *RUBE*, manipulative objects are incorporated in such a way that allows the user to *play* within the *confines* of the model world in much the same way that a child might role-play with toys [8]. This act of playing can contribute immensely to our comprehension of systems and their complexity. Through this kind of hands-on exploration of "possibility spaces" we glean terrific amounts of information about a system [13].

By incorporating interaction, the user also becomes an integral part of the narrative schema and as a result tends to respond *emotionally* to events in the model world. Being tightly coupled with the model world and having a high level of perceived control over its outcome is amazingly parallel to computer gaming, which simultaneously engages emotions of anticipation, joy, and acceptance. Should the model produce unexpected results or behave in an unexpected manner (or even crash) even more emotions are engaged resulting in surprise, anger, and frustration [13]. Coupling these emotional

responses with the presentation of the information has the added benefit of improving comprehension and retention of concepts [3].

Hughes observes that, "Play, in the pure sense of simply controlling and manipulating things, in a safe environment, for no particular purpose, is crucially important for humans."[13:170] He notes that it is often easier to give users this "fun of control" if there is an underlying serious didactic purpose and is surprised to find that educational software is amazingly void of such "virtual toys" that allow for this kind of playfulness and emotional reward [13]. Thus the potential for a system like *RUBE* is great since it engages the user at so many levels and can also create an immersive experience.

Immersion and Engagement

The great potential of a system like *RUBE* for representing mathematical models is that—beyond merely representing systems for communication and analytic purposes—there exists potential for the experience of working with the models to be quite pleasurable. Because *RUBE* is designed to operate in much the same way as interactive narratives and computer games, this pleasure can be both immersive and engaging. The immersive nature of *RUBE* is derived from the way that any given model presentation is derived with a single schema in mind. While interacting with the single model presentation the schema remains constant allowing the user to draw on the scripts commonly associated with that schema to drive perception of the model and the various ways to interact with it [6]. Of course, much of this is dependent on the model presentation design. Interaction with the model should be well guided and should conform to the schema world. Failure to do so will likely have the same result as in other media: frustration [6]. *RUBE* invites users to have several immersive encounters with the models by decoupling model content and presentation. This allows the user to apply

several presentations to the same model, much like a sort of "schema style-sheet". It is also hoped that as users gain understanding of models they will seek out secondary sources to augment their schemas for understanding the models and will then revisit the *RUBE* models to gain greater insight through engaging experiences.

Multi-Sensory Model Representation

Finally, to enhance the immersive experience and provide further language cues, the worlds that are developed for *RUBE* are usually multimedia applications. Humans are quite obviously multimodal beings. Our senses are the mechanisms through which we perceive our world and are integral to our understanding of it. Mixing several types of media has the potential to be "sensory dynamite" [11]. Adding one form of media to another can dramatically change our perception of the original media. Hollywood has used this for years in movies. By laying a soundtrack under the visuals, directors are able to subtly manipulate the mood of a scene and drive audience expectations. Incorporating audio with graphics not only improves the perceived quality of the graphics, but also provides a means of implying information about the graphics and their interactions [11]. In short, the more cues our model representation can give, the more modes of interpretation we can potentially employ; however, Hughes cautions that the use of non-verbal cues should be used subtly. We are highly sensitive to them and misuse (intentional or not) results in being highly intrusive and can very easily have the opposite of the desired effect [11].

Summary

We understand that traditional methods of conceptualizing and representing dynamic systems models abstract the systems to such a level that they are no longer readily comprehendible without formal training in simulation and model design. While

this level of abstraction and formalism is useful for the conceptualization and simulation of the model, we attempt to counter the effects of abstraction on the presentation of the model by using multi-media applications and schema frameworks in our worlds. This approach has a high potential for creating immersive and emotionally engaging experiences when working with the models. This in turn has benefits of improved comprehension and retention of the information presented in the model.

CHAPTER 3
CONCRETIZING CONTENT DEFINITION

Overview

MXL was developed to allow for the formal definition of model topologies and the establishment of simulation frames. The initial version of MXL displayed several weaknesses. Prominent among these were tendencies of allowing ambiguity in model definition and of being unintuitive due to its abstract nature as well as not truly allowing component reuse. These weaknesses necessitated a restructuring of MXL. Since MXL is an XML application, its syntax and grammar are defined using the XML Schema Definition Language (XSD)—also an XML application. Thus, a restructuring of MXL meant refining the MXL schema definition.

Refining the MXL Schema

The current version of MXL was created to address the following issues that were surfaced in the original version:

- Model content and presentation were too tightly coupled.
- Model component granularity was too abstract.
- Subtleties of unique model types were impossible to implement.
- Multi-model nesting rules were over-generalized.
- Model definitions included redundant or superfluous information.
- Simulation frame definition was both incomplete and contained superfluous information.

Table 3-1 provides a summary of the original version of MXL, since that was the starting point used in refining the MXL schema.

Table 3-1. List of original MXL elements with descriptions

Element	Attributes	Children	Description
<MXL>		<model> <simulation>	MXL root element[a]
<model>	type="xsd:string"	<topology> <behavior>	defines topology and behavior of a model
<topology>	type="xsd:string"	<node> <edge>	defines objects and the connectivity of each object in the model
<node>	id="xsd:string" type="xsd:string"	<script> <model>	defines a node in the graph structure of the model
<edge>	id="xsd:string" type="xsd:string" begin="xsd:string" end="xsd:string" data_type="xsd:string"	<script>	defines connectivity between nodes
<script>	id="xsd:string" func="xsd:string"		defines an executable script
<behavior>	type="xsd:string"		defines the type of animation to be attributed to the current node during simulation
<simulation>	type="xsd:string"	<program> <simTime> <inputData>	defines the simulation frame
<program>	name="xsd:string"		defines an executable simulation script
<simTime>	duration="xsd:string"		defines the total simulation time for model execution
<inputData>	type="xsd:string"		defines the source of input data for the simulation

[a] A root element is required in all XML applications

Source: Summary of information found in Kim [14]

The XML schema essentially allows one to specify a context-free grammar for an XML application. I will take a top down approach to explaining the MXL schema, located in Appendix A, and will comment on portions added specifically to address the issues mentioned above and provide code snippets for reference.

MXL Data Types

To disambiguate the data types supported in MXL, we created an enumerated list of data types to be used in all *datatype* attributes of the MXL language. This list is case sensitive, which eases our XSLT processing. The list of data types is based on standard

primitive data types and their array-based counterparts. The names are derived from the Extensible 3D (X3D) language specification. Since raw data is not currently entered directly into MXL files, there was no need to create simple types to complement the enumerated list. However, if the need arises, these types are very nicely paralleled by XSD data types and the array constructs can be created using the XSD *list* element. Table 3-2 summarizes the data types available in MXL.

Table 3-2. MXL data types

Data Type	Description
Boolean	represents a logical type with possible values of (true | false)
Booleans	represents an array of Boolean values
Integer	represents a 32-bit integer value
Integers	represents an array of Integer values
Float	represents a single-precision floating-point value
Floats	represents an array of Float values
Double	represents a double-precision floating-point value
Doubles	represents an array of Double values
String	represents a character string type
Strings	represents an array of String values

The enumerated list makes it simple to modify the data types we support as well. By adding or removing an XSD *enumeration* element, the corresponding data type is added or removed to all attributes of the MXL language that use the enumerated list to define type, respectively.

MXL Root and Top-Level Elements

Figure 3-1 shows the portion of the schema that defines the MXL top-level elements. These elements are discussed in detail the sections that follow.

The MXL root element

The MXL root element remains basically the same as its original form. The only change that has been made is the addition of a target namespace attribute with a resource locator so that developers may validate their MXL files against the MXL schema

definition. This namespace information will eventually be redirected to the main *RUBE* website at http://www.cise.ufl.edu/~fishwick/rube.

```
<xsd:schema targetNamespace="http://www.cise.ufl.edu/~kdamkjer/rube/MXL"
xmlns="http://www.cise.ufl.edu/~kdamkjer/rube/MXL" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
elementFormDefault="unqualified" attributeFormDefault="unqualified">
…
  <xsd:element name="MXL">
    <xsd:annotation>
      <xsd:appinfo>The MXL file is broken into two main sections: Model Definition and Simulation
Frame.</xsd:appinfo>
      <xsd:documentation>This is the MXL root node.</xsd:documentation>
    </xsd:annotation>
    <xsd:complexType name="mxlType">
      <xsd:sequence>
        <xsd:group ref="models"/>
        <xsd:element name="simulation" minOccurs="0">
          <xsd:complexType name="simulationType">
            <xsd:attribute name="start_time" type="xsd:float" use="required"/>
            <xsd:attribute name="end_time" type="xsd:float" use="required"/>
            <xsd:attribute name="delta_time" type="xsd:float" use="required"/>
            <xsd:attribute name="cycle_time" type="xsd:float" use="required"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
…
```

Figure 3-1. MXL top-level elements

The *models* group

The *model* element has been replaced by a group structure. This means that the *model* element no longer appears in the MXL file, though it is still present in the schema definition as the logical construct *models*. This group serves to encapsulate all currently supported model categories (also groups) and to establish the basic model interface. By handling models in this manner we are able to implement the subtleties of unique model types without introducing ambiguities and at the same time provide a consistent framework in which the model types are defined.

The *simulation* element

The *simulation* element remains, though it has been substantially revised. All child elements of *simulation* have been removed, making *simulation* a leaf element. The only

child that still exists in some form is the *simTime* element which has been replaced by several "time" attributes that serve to establish the simulation frame: *start_time*, *end_time*, *delta_time*, and *cycle_time*. The *program* and *inputData* elements were removed since the aspects of the model that they were attempting to capture were better defined elsewhere. Input data is more clearly defined as being fed from a super-model or generator block. The program script that the *program* element referred to is actually the script that is created by the MXL-to-DXL-to-Script portion of the *RUBE* merge pipeline.

Removal of the *comment* element

The *comment* element[1] was removed since the XML comment style suffices despite the fact that XML parsers can legally ignore these comments. Should we find that this element is again needed (to ensure that comments survive processes such as XSLT) we will add the element back in, or provide a similar method for annotating MXL files.

Model Definition Elements and Groupings

The new *models* group contains two sub-groups comprising of the model categories and the model interface. The *topology* and *behavior* elements of the previous version have been removed. It was found that model topology was inherent in the unique model types and that specifying it was superfluous. The *behavior* element was removed since it coupled model presentation with model content definition too tightly.

The members of the model interface *handles* group, *input* and *output*, were added to provide a consistent method of explicitly establishing the type of information to be sent and/or received by a model. The previous version attached these attributes to edges. Through experimentation we tested modifying how the attributes were attached to edges

[1] Not included in Table 3-1 since it was never officially documented, but I mention it since it was in the original schema definition.

and attempted attaching the attributes to nodes, but we ultimately came to the conclusion that neither method would work without feeling artificial. The handles provide a definite and consistent structure for specifying model input, output, and a data type for each without having to completely rework the interface for each new model type.

```
…
  <xsd:group name="models">
    <xsd:annotation>
      <xsd:appinfo>All model categories should be placed under the "choice" element.  New model
categories should be appended to the end of the current list.</xsd:appinfo>
      <xsd:documentation>Generic Model Group</xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:group ref="handles"/>
      <xsd:choice>
        <xsd:group ref="functionalModels"/>
        <xsd:group ref="declarativeModels"/>
        <xsd:group ref="scripts"/>
      </xsd:choice>
    </xsd:sequence>
  </xsd:group>
…
```

Figure 3-2. The *models* group

Handles: *input* and *output*

```
…
  <xsd:group name="handles">
    <xsd:annotation>
      <xsd:documentation>Handle Group</xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="input" minOccurs="0" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:attribute name="id" type="xsd:ID" use="required"/>
          <xsd:attribute name="datatype" type="fieldTypeName" use="required"/>
          <xsd:attribute name="initial" type="xsd:NMTOKEN" use="optional"/>
          <xsd:attribute name="index" type="xsd:unsignedInt" use="required"/>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="output" minOccurs="0" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:attribute name="id" type="xsd:ID" use="required"/>
          <xsd:attribute name="datatype" type="fieldTypeName" use="required"/>
          <xsd:attribute name="index" type="xsd:unsignedInt" use="required"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:group>
…
```

Figure 3-3. The *handles* group

An emergent property of adding the interface handles was the ability to immediately verify that models and model components nested and joined properly. Though this verification capability has not yet been implemented in the *RUBE* framework, visual checks are now much easier to perform and an automated process of checking the congruency of the number, type, and ordering of conjoining handles should be fairly easy to implement.

Model categories

Figure 3-2 also illustrates the method that is employed for handling different model *types*. Model categories, or families, are created under the generic model group. Several potential model families include conceptual, constraint, declarative, equation-based, functional, spatial, and stochastic. Under these family groupings, specific model types and their components are defined. It is here that nuances of specific model topologies should be exposed.

At present, MXL implements declarative and functional types directly, and it is hoped that we will also be able to support equation-based types using the Mathematics Markup Language (MathML) XML application in the near future. MXL currently contains definitions for the following functional models: Functional Block Models (FBM), System Dynamics Models (SDM), and Script-based Models (Script), as well as the following declarative models: Finite State Machines (FSM) and Petri Networks (P-Net). It should be noted, however, that only FBM, FSM, and Script have been tested extensively and are fully implemented in the *RUBE* framework.

Generalizing Node Definition Structure

As mentioned previously, Scripts are considered a type of functional model. The *script* element also serves as the only current leaf node in the MXL model definition,

which is why it is the vehicle used for defining model semantics. Just as we want to be able to reuse sub-models to create more complex models, we also want to be able to build

Table 3-3. MXL model elements

Element	Attributes	Children	Description
<script>	lang="xsd:string {Java | JavaScript}" src="xsd:anyURI" func="xsd:NCName"	<param>	defines a code-based model in the specified language
<param>	value="xsd:NMTOKEN" index="xsd:unsignedInt"		defines a parameter to pass to the script function
<fbm>	id="xsd:ID" src="xsd:anyURI"	<block> <trace>	defines an FBM
<block>	id="xsd:ID"	{models}	defines an FBM block node
<trace>	from="xsd:IDREF" to="xsd:IDREF"		used to connect FBM blocks
<sdm>	id="xsd:ID" src="xsd:anyURI"	<src> <sink> <rate> <level> <aux> <const> <arc>	defines an SDM
<src>	id="xsd:ID"		defines an SDM source node
<sink>	id="xsd:ID"		defines an SDM sink node
<rate>	id="xsd:ID"	{models}	defines an SDM rate node
<level>	id="xsd:ID" intial="xsd:double"	{models}	defines an SDM level node
<aux>	id="xsd:ID"	{models}	defines an SDM auxiliary
<const>	id="xsd:ID" value="xsd:double"		defines an SDM constant
<arc>	type="xsd:string {flow | cause-effect}" from="xsd:IDREF" to="xsd:IDREF"		used to connect SDM nodes
<fsm>	id="xsd:ID" src="xsd:anyURI"	<state> <transition>	defines an FSM
<state>	id="xsd:ID" start="xsd:boolean	{models}	defines an FSM state
<transition>	from="xsd:IDREF" to="xsd:IDREF"	{models}	used to connect FSM states
<p-net>	id="xsd:ID" src="xsd:anyURI"	<condition> <event> <connection>	defines a P-Net
<condition>	id="xsd:ID"	<token> {models}	defines a P-Net condition
<token>	color="xsd:nonNegativeInteger" init_tokens="xsd:nonNegativeInteger"		used to initialize P-Net tokens
<event>	id="xsd:ID" time="xsd:double"	{models}	defines a P-Net event
<connection>	from="xsd:IDREF" to="xsd:IDREF"	{models}	used to connect P-Net nodes

libraries of model semantic components. In the previous version of MXL, this was not possible since it relied heavily on name correspondence to establish a mapping between model structure and semantics. To add this functionality, which is essential for not only building semantic libraries, but for multi-modeling as well, we added the *index* attribute to our model handles. This abstracts the model to a more generalized structure that is not dependant on its parent. For sub-models, we simply require that the handle type, number, and ordering match. For program scripts, we create a generalized input array and output array to represent the MXL model handles. We also allow the user to specify any parameters that should be passed as well, which is occasionally necessary for components such as constant generator blocks. This generalized structure allows retrieving input and sending output to simply become an array look-up based on the handle's *index* attribute.

CHAPTER 4
EXTENDING *RUBE* TO SUPPORT THE MULTI-MODELING PARADIGM

MXL External Model File Referencing

Added support for multi-modeling was among the several changes to MXL. In the previous version of MXL, the entire model definition had to be included in a single large file. This meant that self-similar model sub-components had to be written repeatedly instead of being written once and then referred to by the parent model. This increased the likelihood of typographical errors being introduced into complex models and made debugging terribly difficult. For extremely complex or large models, the model definitions became incomprehensible and indecipherable.

To address this issue, the *src*[1] attribute was introduced on all of our model root elements and it is recommended that it be included on all future model root elements. This attribute contains a Universal Resource Identifier (URI) reference to an external model definition. This allows developers to deal with models in manageable chunks, drastically reduces file size, improves readability, and increases development efficiency. Since developers need only create one model definition for a sub-model, the chance for error decreases. Similarly, should errors be identified in model sub-components, they now need to be changed in only one location and changes propagate throughout the models that refer to the sub-model.

[1] *src* is an abbreviation of "source" that was chosen for mnemonic reasons. Its functionality should be immediately apparent to those familiar with HTML or similar mark-up languages.

The model interface handles are employed as the means of communication between parent and child models. Consider Figures 4-1 and 4-2, code snippets of a parent model and its corresponding, externally defined, sub-model component.

```
…
    <block id="myBlock">
      <input id="Input_01" datatype="Double" index="0"/>
      <input id="Input_02" datatype="String" index="1"/>
      <input id="Input_03" datatype="Boolean" index="2"/>
      <output id="Output_01" datatype="Double" index="0"/>
      <fbm src="./mySubModel.xml"/>
    </block>
…
```

Figure 4-1. Parent model external reference

```
<?xml version="1.0" encoding="UTF-8"?>
<mxl:MXL xmlns:mxl="http://www.cise.ufl.edu/~kdamkjer/rube/MXL"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.cise.ufl.edu/~kdamkjer/rube/MXL
http://www.cise.ufl.edu/~kdamkjer/rube/MXL/MXL.xsd">
  <input id="externalInput_01" datatype="Double" index="0"/>
  <input id="externalInput_02" datatype="String" index="1"/>
  <input id="externalInput_03" datatype="Boolean" index="2"/>
  <output id="externalOutput_01" datatype="Double" index="0"/>
  <fbm id="myFBM">
    …
  </fbm>
</mxl:MXL>
```

Figure 4-2. Externally defined sub-model (mySubModel.xml)

The two parts fit together like puzzle pieces. The rule that is used to ensure a valid sub-component substitution is that the number, type, and index of the corresponding input and output handles must match. In the above example both referrer and referee models have the same number of input and output handles, the input index “0” on both models is of type “Double”, the input index “1” on both models is of type “String”, and so forth. Note that name correspondence plays no role in mapping parent structure to child structure. This allows model components to be reused several times in a single model.

MXL Massive Merge

To incorporate these changes into the *RUBE* framework a slight adjustment to the *RUBE* pipeline was needed. In the *RUBE* pipeline, MXL—a high-level development language—is translated into the Dynamic Exchange Language (DXL)—a lower-level assembly-type language—via XSLT and is further used to generated hooks to the simulation data in the target presentation language.[2] To successfully perform this translation, the disparate MXL files must first be joined into a cohesive whole. This process is the "MXL massive merge" and is incorporated into the existing MXL2DXL.xsl that contains the rules for the MXL-to-DXL translation.

The single massive MXL file is constructed from the model sub-components by substituting model definitions from referred files for the parent model references. These substitutions take place recursively during a top-down descent of the base MXL file and its children, thus several levels of multi-model hierarchy are easily attainable. Of course, a few issues arise in handling the merge process, specifically: element identifier conflicts and joining interface handles of parent and child models.

Resolving identifier conflicts is achieved by using a simplistic, yet seemingly effective algorithm for creating new identifiers for the child models by appending an underscore character[3] to the parent containing element's identifier and further appending the child identifiers to this construct. This algorithm is applied recursively resulting in name-chains that reflect the nesting hierarchy of the model.

---

[2] The target presentation language is usually VRML, though translation mechanisms for other target languages, such as SVG, are being developed.

[3] This character is added to improve the readability of the, often long, composite names.

Similarly, joining interface handles is achieved by an equally simplistic algorithm. The handles for the parent referrer model are kept[4] and identifier references that refer to the child handles in any connection elements of the child model are replaced with the parent identifiers using the *index* attribute as a mapping mechanism.

Model Libraries

With this improved handling of the multi-modeling paradigm, developing libraries of commonly used models is fairly simple. There is currently a consideration to develop common libraries for basic mathematical functions for distribution with the *RUBE* framework. Including such libraries would reintroduce an aspect of prior modeling and simulation toolkits that has been absent from the *RUBE* Project and would further aid model developers. Regardless, it is now easy enough for developers to implement their own custom libraries and easier for them to share those libraries with the modeling and simulation community.

[4] Parent handle identifiers may potentially be modified due to the renaming algorithm; this has been accounted for in that the handle replacement algorithm occurs *after* the renaming algorithm.

CHAPTER 5
MODEL DEFINITION

Overview

Given that models, by definition, are abstractions of the real-world phenomena that they describe which capture specific attributes of the phenomena being modeled, then for any given system there are potentially several valid ways to construct a model representing that system. Each model can be correct in its own right, and intent plays a large role in determining the validity of the model with respect to a particular application. As such, this chapter seeks to provide general guidelines for producing concise, well-defined models that take advantage of the *RUBE* framework's support of the multi-modeling paradigm.

When designing models, it is recommended that the developer diagram and describe the model since this can greatly aid in identifying several of the trends listed below.

Identification of Model Subcomponents

Recall that the multi-modeling paradigm incorporates any or all of the following in a model design: *multiplicity*, *heterogeneity*, and *hierarchy*. As such, notice that even single level models can be considered multi-models and that defining models concisely and efficiently is the result of being able to identify these trends in a model's design.

Self-Similarity of Model Functionality and Interface

One of the most basic approaches to identifying model sub-components is to use self-similarity of model component functionality and interface as a guideline. This test

surfaces model *multiplicity* in model design and definition. Recall that the basic test for determining if a referenced child model can replace a parent reference is to ensure that the interfaces match by comparing the number, type, and ordering of the model interface handles. A similar test could be used to determine the interchangeability of coexistent model components. If there are multiple model components that exhibit identical interfaces, it is probably worth investigating the similarity of the models' functionalities. If the model functionalities are similar, then the developer can likely create a single sub-model component to encapsulate the specific model semantics.

Note that the functionality need only be *similar*, not *identical*. Recall that the *script* element can contain *param* elements that can serve to provide slightly different functionality from the script-based model. One such popular use is in FBM constant-generator blocks where the *param* element is used to specify the constant desired and the Script code merely echoes the parameter as its output. Another possible use, also for FBM generator blocks, is to use several *param* elements to determine the start-range, end-range, seed, and distribution-type for a generic random number generator.

Logical and/or Functional Groupings

Another approach, which is useful for surfacing *hierarchy* in model design and definition, is to search for logical and/or functional groupings in the model design. These groupings can be defined as sub-model components. This allows the parent model to remove the definition of the group and place in its stead an abstract representation that encapsulates the group. These regions often appear several times in a complex model, and consolidating the group definitions aids not only in reducing complexity at any given model level, but also in later troubleshooting of the model definition and semantics. Note that this approach can be used repeatedly to create groups of groups, that is, several

layers of model hierarchy. Also, this approach is just as applicable to declarative models, like FSMs, as it is to functional models; creating hierarchical FSMs is actually quite useful in representing complex state-spaces.

Object-Oriented Approach

Thinking of model components in terms of the object-oriented paradigm often surfaces *hierarchical* relationships and provides a framework for incorporating *heterogenic* model types in a complex model definition. These *hierarchical* relationships can be reflected in the model definition by using the model reference capability to refer to various model components. Incorporating *heterogenic* model types becomes trivial since MXL, like the object-oriented paradigm, provides a common interface to communicating between disparate model types. Since model definitions are encapsulated, parent models need not know anything about the implementation details of the child models. It should be noted that some tenants of the object-oriented paradigm are not implemented in MXL. Among those missing, *inheritance* is probably most apparent; there is no sharing of information between parent and child outside of the information that is explicitly passed through the interface handles. It has been suggested that scoping capabilities in MXL and the *RUBE* framework be explored, but this research has yet to be completed. The other tenant that is currently unsupported is *polymorphism*.

Optimizing Model Definition

It is worth making a brief note about model optimizations. Certain combinations of models have the potential to exhibit "distributive" properties. The most common of these is FBMs nested within the states of a FSM. It has been observed that model developers will occasionally use this construct as a sort of model switch. While this structure is perfectly legal in terms of FSM semantics—describing state-spaces and the means for

transitioning between them—it can be greatly optimized if instead of nesting identical FBM structures in the FSM states, a single FBM that receives state information from the FSM is used.

Since non script-based models are translated into a generalized event scheduling and queuing model for simulation, performance can be improved by providing script-based definitions of sub-models instead of MXL definitions. Though a certain degree of the model semantics is abstracted away, this solution can be useful for those attempting to improve simulation performance.

CHAPTER 6
METAPHORS TO MODEL BY

One of the main goals of the *RUBE* Project is to improve model comprehension by using metaphorical representations of model structure and execution. These metaphors seek to operate within the confines of schemas to frame and direct user expectations while interacting with the model world. However, appropriate metaphors to use in these presentations have remained a largely unexplored area.

Metaphors are a somewhat difficult beast to tame since the only way to explain them is through other metaphors (note the use of metaphor in this very sentence: "problems are wild animals"). The very names that have been assigned to the model structures are themselves metaphors for the model content that we are seeking to define. Metaphor is so tightly ingrained in our lives that we hardly ever notice we are using it. This is both the beauty and the bane of metaphor: when used well, we accept an explanation because it fits with preconceived notions; when forced, we reject an explanation because it exhibits artificiality.

This chapter seeks to summarize some of the prominently recurring and successful metaphors. If you pay close attention to the language used to converse about modeling and simulation, several of these metaphors should seem obvious in retrospect since you were probably already using them unconsciously. In fact, several of these metaphors seem to be related to Lakoff's "event structure" and "mental events" metaphors [15]. I will point out correlations where appropriate.

Model Structure

The vast majority of the model types used in modeling and simulation are graph-based structures. This is reflected in the categories of metaphorical mappings found in the presentation of these models. The major structural categories focus on the following aspects of model structure: models and sub-models, model interfaces, nodes, and edges.

Model Metaphors

Models are formalized conceptual constructs, whether they are tightly coupled to a real-world physical system or a more abstract structure. As such, it stands to reason that metaphors for "ideas" are particularly useful for mapping onto models for presentation purposes.

Models are environments

"Models are environments" is related to Lakoff's metaphors: "ideas are locations" and "subjects are areas" [15]. We tend to refer to thoughts and ideas as landscapes that we move through when attempting to comprehend or explain concepts. This metaphor is particularly effective for use in model presentations. In fact we tend to think of the root model as being the "world" structure in most presentations. Similarly, sub-models can be represented as regions or microcosms within the larger, more encompassing, parent structure.

Models are containers

Another popular metaphor for models is, "models are containers." This metaphor can be a logical continuation of the previous metaphor since one can consider the "world" or "environment" as a sort of *super-container*. This metaphor, however, seems to imply that noticeable boundaries are established. It is used almost exclusively when diagramming models. Models tend to be drawn as *boxes* with components being placed

with-in these model-containers. This metaphor also appears consistently in the model presentation development methodology presented in Chapter 7.

Models are buildings

The "models are buildings" metaphor is possibly linked to Lakoff's "states are places" metaphor [15]. It can also be seen as an extension of the "models are containers" metaphor since buildings are large specialized containers. This can be a particularly powerful metaphor since the function of an organization or the activities that occur within a building are often reflected by its architectural appearance.

Models are machines

"Models are machines" represents models as independently functioning machines that work in tandem towards a larger purpose. Each model in the system serves a specific specialized role and by combining models together into more complex models and model units, emergent properties are exhibited. This metaphor works well when used in conjunction with the "models are buildings" or the "models are containers" metaphors when designing multi-models.

Handle Metaphors

Handles are the interfaces for models. They are the means by which data enters and leaves models and they are the means by which models are joined. This behavior is reflected in the metaphors used to represent these constructs.

Handles are thresholds

"Handles are thresholds" treats the model interface as a doorway, or boundary that must be crossed in order to interact with the model. It reflects the view that the interface is the means by which data enters and leaves the model system. An extension of this

metaphor is that handles are secure or guarded doorways, allowing access to the model only to the right types of data.

Handles are valves

"Handles are valves," similar to the guarded doorway, treats the model interface as a controlling structure. This metaphor tends to be used in conjunction with the "models are containers" metaphor; the valves control the flow of information into and out of the model. Also, differently sized pipes, or different types of fluid are typically used to represent the different types of information when used with this metaphor.

Handles are ports or outlets

"Handles are ports or outlets" views the interface as "something to be plugged in to." This is an interesting metaphor to use if you consider computer ports where the interface is actually a composite of several smaller ports. The complete resulting interface requires an appropriately shaped jack with the proper "pin out" to communicate.

Handles are radio towers

"Handles are radio towers" treats the models as transmitting and receiving devices. The towers tend to operate on specific frequencies and amplitudes and the towers can contain several transmitting and receiving dishes. This metaphor can be especially applicable if signal degradation is a factor in your simulation since most people have experienced the radio signal breaking up as they venture outside of a station's broadcast area. This metaphor is also extremely applicable in models that operate asynchronously.

Handles are puzzle pieces

Thinking of interfaces as puzzle pieces is representative of a conceptualization of compositing smaller, seemingly insignificant models, into a large comprehensive model. This metaphor reflects the view that the model interface is the means by which models

are joined. This metaphor does exhibit some very specific drawbacks though in that *joining* is the extent of the interface's functionality that is represented. A possible extension to "smart" puzzle-pieces, possibly integrated circuits, may make this a more useful metaphor.

Handles are fasteners

Similarly, representing handles as fasteners surfaces the joining properties of model handles, yet again, displays the same drawback as the "puzzle piece" metaphor.

Node Metaphors

The difference between models, nodes, and sub-models is often very fuzzy. As a result the metaphors used to represent nodes are usually very similar to those that are used to represent models. Though the metaphors are nearly identical, there is usually a degree of separation where the model acts as the container for the nodes. Note that remembering the scope of your presentation world can be very important. If the model is represented by a *city*, the nodes are *buildings* with the city. However, if the model is represented by a *building*, nodes are *rooms* within the building. Further, if the model is represented by a *room*, the nodes are represented by *objects* within the room, and so forth. Also note that this architectural metaphor is one of several popular multi-modeling presentation schemas.

Nodes are containers

The general metaphor, "nodes are containers," seems to be safe to use almost anywhere in the model hierarchy. Excessive use can make for very unintuitive presentations since the metaphorical mapping will consist of abstract concepts mapped onto generic "things."

Nodes are rooms

As mentioned above, “nodes are rooms” is used in conjunction with the “models are buildings” metaphor. The architectural metaphor can be very useful in models that have an extreme amount of hierarchy since buildings can be constructed from several layers: rooms compose offices, which compose sections, which compose floors, which compose buildings, which compose complexes, seemingly *ad infinitum* (or perhaps more appropriately, *ad nauseam*).

Nodes are machine parts

“Nodes are machine parts” is an extension of the “models are machines” metaphor. In this example, the nodes are the functional cogs and inner workings that drive the model and make it work. It should be noted that analog devices tend to be much more effective since they display a higher degree of animism on model simulation. This makes it much easier to attach causal behaviors to the model components. Digital metaphors tend to be much more stagnant representations which can make models difficult to comprehend. For example, consider opening a digital wristwatch and a mechanical wristwatch to reveal their inner workings. In which would it be easier to immediately understand and see causal relationships between the subcomponents?

Edge Metaphors

Edges are the connecting elements of our model structures. They are the most consistently represented model element because they are nearly always represented by conduits. On occasion, a presentation developer may use special orientation to infer edge-connections, but these representations tend to be harder to comprehend than those that use concrete conduits.

Edges are conduits

"Edges are conduits" is a fairly general all-encompassing metaphor to use with edges. Conduits come in many varieties. The most common conduits that are used in model presentations are as follows: paths and roads, pipes, wires, ducts, and abutting passageways. There is another, less frequently used, representation: waves. In this example the conduit is usually atmosphere or fluid and is generally only used when broadcasting information is an integral metaphor to the phenomena being modeled.

Edges are a line-of-sight

On very rare occasions model developers will use VRML *Viewpoint* nodes to indicate the current model state. In these cases, it has been observed that the developer uses line-of-sight to indicate reachable next-states. This metaphor tends to be highly specialized and should be used cautiously since presentation ambiguities almost certainly accompany its use. One suggestion is to use this metaphor in combination with a concrete one; this may serve to augment the presentation and increase user immersion.

Model Execution

Simulation is generally characterized by activity, animation, and animism in presentations; thus, most of these metaphors center on activity and spatial-temporal relationships. Simulation metaphors tend to focus on the state of model during execution, flow of data through the model, and on model (and sub-model) input and output.

Input Metaphors

We commonly think of input as a driving force. Its introduction into a system creates causal relations among the systems components. Hence, several of these metaphors are tightly coupled with Lakoff's "causation" metaphors [15].

Input is fuel

The metaphor of "input is fuel" surfaces several conceptions about input and can be quite effective depending on the schema being employed. This metaphor indicates that we often think of input as being sporadically supplied, as opposed to coming from a constant stream; that model performance is highly coupled with the quality of the input received, that input is a driving force or power; and that the input itself is often changed in the course of model execution.

Input is a source

Unlike the previous metaphor, the "input is a source" metaphor views input as a wellspring, continually and consistently supplying data for processing. Note that source here refers to a water source, not a generic "location of origin."

Input is an agent entering

The "agent" or "avatar" metaphor for data is complex. In this metaphor, an avatar or user agent represents a chunk of information that progresses through a system. This means that the agent takes on various roles throughout the course of the model execution. While a very effective metaphor, it is important to understand the various roles that the agent plays throughout model execution. "Input" in represented in a couple of ways. For an architectural schema, input is generally represented by the agent entering, or crossing-over into a model.

Input is a message

Similarly, input can be viewed as a message delivery. This message can be delivered straight to the model, (as in a transmission received by a machine) or it can be delivered from a user agent (used in tandem with the "agent entering" metaphor). This

metaphor is good for using with the agent construct while the agent is moving within a local model.

Input is an action

The "action" metaphor treats input as in impulsive force. This metaphor is usually employed when thinking of models as "black boxes." We supply input with the intent of receiving output and have no interest in the mechanics that produce the output. This metaphor is actually one part of a two part "action-reaction" metaphor and should be used accordingly.

Data Flow Metaphors

Models are often employed to visualize and analyze the optimality of a system. This analysis requires observing data as it moves through and is manipulated by the system; this is referred to as data flow.

Data flow is a transfer of energy

This metaphor can be related to the "input is action" metaphor. In this case, data flow is represented as a transfer of energy. This can be visualized in terms of colliding objects or force applied to objects over time. Observe that the transfer of energy need not be acute; that is, that the transfer of energy can take place smoothly over time.

Data flow is flowing water

The immediate metaphor of "data flow is flowing water" should be obvious. When analyzing systems we are often looking for "bottlenecks," "leaks," "overflows," and other such liquid-based names for problems and restrictions. This metaphor works well with the "input is a source" metaphor and "models are containers" metaphor.

Data flow is a moving agent

This is a continuation of the "agent" metaphor. Data flow, or transferring of information, is represented in the "agent" metaphor by moving the agent through the system along paths or through conduits so that it may interact with various model components.

Data flow is a transmission

This metaphor is best used with the "handles are radio towers" metaphor. It is representative of the means of transferring information between towers. Again, it is most effective when broadcasting information is an integral part of the model.

Active Node Metaphors

While the model is executing, it is often helpful to know the current state of the system. This is where "active nodes" are useful. Marking the active nodes helps provide a visual summary of the model state. This can be a powerful analysis tool if declarative model types are used since these models define state spaces and the transitions between them. Not all state spaces are necessarily valid, and being able to identify if or when a model is in an invalid state can be very important in these circumstances. Because of this desired behavior, active states are usually represented in such a way as to stand out from the rest of the model.

Active node is animation

One popular method of demarking active state is by animating the current state in some way. It should be noted that animation need not involve transforming the model object (which could be problematic in architectural schemas). Swapping Image Textures for Movie Textures, giving the appearance of lights illuminating, etc. are all excellent

ways of indicating current states. It is critical that schema be considered when choosing these metaphorical mappings.

Active state is agent location

In the "agent" metaphor, active states are simply the locations in which the agents are present.

Active state is a color or texture

"Active state is a color or texture" is along the same vein as "active node is animation." Changing material properties can be very effective at making model elements stand out. A cautionary note when using this metaphor: used in extremes it can give object the appearance of undergoing alchemical transformations, which may be confusing to observers.

Active node is a sound

"Active node is a sound" is one of the best metaphors for indicating active state. Auditory cues are usually underutilized in model presentations. The only warning: make the sound clips used disparate enough to be distinguishable, yet somehow related enough to work well together (or at least function transparently) under the selected schema.

Output Metaphors

Output is the product of model execution. It is result of the model functioning. The metaphors used to represent output tend to treat output as a tangible object. Several of these metaphors are direct complements to their "input" counterparts.

Output is a manufactured product

"Output is a manufactured product" is used typically where models are represented as factories (a specialized building that contains production machinery). It surfaces the "produced" aspect of model output.

Output is a sound

The "output is a sound" metaphor is the big exception to the "output represented as a tangible object" tendency. In this case the result of the model functioning is auditory. Consider instruments as special forms of mechanical devices that take as input some form of excitation and output auditory information. In this specialized case, the model executing over time produces music.[1]

Output is an agent exiting

"Output is an agent exiting" is the complement of "input is agent entering."

Output is a reply

"Output is a reply" is the complement of "input is a message."

Output is a reaction

"Output is a reaction" is the complement of "input is an action."

Output is a piece of art

The "output is a piece of art" metaphor is analogous to the "output is a manufactured product" metaphor, though the output visualizations generally have a more aesthetic appeal to them when using the metaphor. Output in these cases is usually a 2D drawing or 3D virtual sculpture, though other art forms can be mapped as well. There is currently research to map literary narrative structure onto models and I have seen simple "dance" or "demonstrations" that were driven by declarative models.

[1] The aesthetic quality of this music may indeed be debatable, though some decent "sounding" audio has been produced using cellular automata, a form of declarative modeling, and harmonic frequencies.

CHAPTER 7
MODEL PRESENTATION DESIGN METHODOLOGY

In this chapter I will explain the methodology for moving from model diagrams and definition to a virtual presentation. Recall that the primary presentation language is VRML; I will be using this as my target language in these examples. For this example I am using a fairly simple FBM from Fishwick [8; 148]. The FBM is diagrammed in Figure 7-1 with MXL definition following in Figure 7-2. Though the methodology outlined in this chapter is explained using a simple example, it can be easily extended to much more complex models as will be demonstrated in the following case-studies.

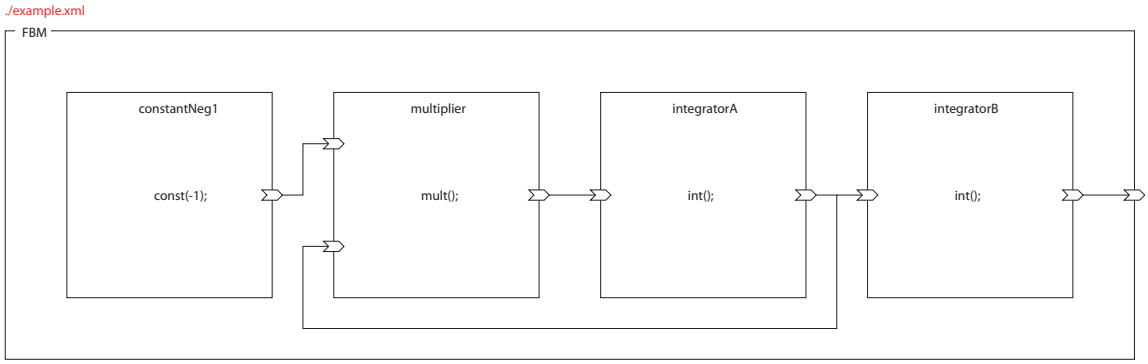

Figure 7-1. Simple FBM block diagram

Model Prototypes

Referring to Figures 7-1 and 7-2, it should be observed that even in such a simple model, elements of multi-modeling are already apparent. The two integrator blocks are examples of model instances. To ensure consistent representations of these identical model components, Prototypes should be used. As the name suggests, by using Prototypes we can define model presentation once and reuse it several times. Prototypes

```
<?xml version="1.0" encoding="UTF-8"?>
<mxl:MXL xmlns:mxl="http://www.cise.ufl.edu/~kdamkjer/rube/MXL"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.cise.ufl.edu/~kdamkjer/rube/MXL
http://www.cise.ufl.edu/~kdamkjer/rube/MXL/MXL.xsd">

  <!-- The model on page 148 contains an external output, so it is included here. -->
  <output id="X" datatype="Double" index="0"/>

  <!-- This FBM is based on the diagram on page 148 of Simulation Model Design and Execution. -->
  <fbm id="myFBM1">

    <!-- Blocks are defined in the same order as in the book.  Block 0 is a constant: -1 -->
    <block id="constantNeg1">
      <output id="Neg1" datatype="Double" index="0"/>
      <script lang="JavaScript" src="math.js" func="constant">
        <param value="-1" index="0"/>
      </script>
    </block>

    <!-- Block 1 is a multiplier -->
    <block id="multiplier">
      <input id="FactA" datatype="Double" index="0"/>
      <input id="FactB" datatype="Double" index="1"/>
      <output id="Prod" datatype="Double" index="0"/>
      <script lang="JavaScript" src="math.js" func="mult"/>
    </block>

    <!-- Block 2 is an integrator -->
    <block id="integratorA">
      <input id="integrandA" datatype="Double" index="0"/>
      <output id="integralA" datatype="Double" index="0" initial="1.0"/>
      <script lang="JavaScript" src="math.js" func="int"/>
    </block>

    <!-- Block 3 is an integrator -->
    <block id="integratorB">
      <input id="integrandB" datatype="Double" index="0"/>
      <output id="integralB" datatype="Double" index="0" initial="1.0"/>
      <script lang="JavaScript" src="math.js" func="int"/>
    </block>

    <!-- Once Blocks have been defined, we need to connect them. -->
    <trace from="Neg1" to="FactA"/>
    <trace from="Prod" to="integrandA"/>
    <trace from="integralA" to="integrandB"/>
    <trace from="integralA" to="FactB"/>
    <trace from="integralB" to="X"/>
  </fbm>

  <!-- Set the simulation to run from time 0 to time 50 with a time slice of 0.01 at a target "frame rate" of 30fps --
>
  <simulation start_time="0" end_time="50" delta_time="0.01" cycle_time="0.0334"/>
</mxl:MXL>
```

Figure 7-2. Simple FBM MXL definition

have a distinct advantage over simply using VRML *Include* nodes in that the prototype interface provides a means of surfacing attributes of the sub-model components. This is

important for routing simulation data to the scene graph for animation during simulation. When developing model presentations, I usually leave these interfaces blank until I know what attributes need to be surfaced. Figure 7-3 provides the basic skeletal structure for the example presentation file which includes a VRML *PROTO* node for each model.[1] Note that we place the root model in a *PROTO* node as well and then instance it. This allows us to later reference this file via an *EXTERNPROTO* node in a larger presentation file, in which case, only the *PROTO* node definitions are used.

```
#VRML V2.0 utf8

PROTO Constant [] {
}

PROTO Multiplier [] {
}

PROTO Integrator [] {
}

PROTO p148 [] {
}

DEF p148 p148 {}
```

Figure 7-3. A VRML presentation file showing model prototypes

Using Groups and Transforms to Mimic Content Structure

Logical Groups

Organization is very important in ensuring consistency of the model presentation with the MXL model definition. Recall that the MXL schema was designed with logical groupings; it makes sense to duplicate these structures in the model presentation file for increased clarity. VRML group nodes should be used in each model prototype definition to provide a logical structure for the model presentation. For each model type there are major groupings of “handles” and “model”, and minor groupings of “input” and “output”

[1] Recall the scripts are considered a type of functional model.

for handles, and “nodes” and “edges” for models. Figure 7-4 shows these grouping structures in place for the root-model prototype. The grouping structures are similar for each of the remaining *PROTO* nodes. Note the naming convention; it helps to be consistent. I use the *PROTO* node type followed by an underscore and the major or minor grouping.

```
…
PROTO p148 [] {
  DEF p148_Handles Group {
    children [
      DEF p148_Outputs Group {
        children [
        ]
      }
    ]
  }

  DEF p148_Model Group {
    children [
      DEF p148_Nodes Group {
        children [
        ]
      }

      DEF p148_Edges Group {
        children [
        ]
      }
    ]
  }
}
…
```

Figure 7-4. Model component grouping in presentation files

Containers

A structure that is not present in MXL, but is extremely useful for model presentations, is the containing transform. I highly recommend using containing transforms on all sub-model component geometry to ease adding model behavior for simulation. Doing so resets the local coordinate system of the contained geometry to the origin, which means that the developer spends less time trying to convert transformations to a world coordinate system; all transformations become relative to the containing

transform. This is extremely useful, especially in multi-model presentations where there may be several layers of sub-model components. This also makes positioning sub-model components in the model world much easier since moving the transform moves *everything* nested beneath it. Figure 7-5 shows these structures nested beneath their respective group structures for the root-model. Notice that I use them on *all* sub-model components.

```
…
PROTO p148 [] {
  DEF p148_Handles Group {
    children [
      DEF p148_Outputs Group {
        children [
          DEF p148_X_Container Transform {
            children [
            ]
          }
        ]
      }
    ]
  }

  DEF p148_Model Group {
    children [
      DEF p148_Nodes Group {
        children [
          DEF p148_Constant_Container Transform {
            children [
            ]
          }
          DEF p148_Multiplier_Container Transform {
            children [
            ]
          }

          DEF p148_IntegratorA_Container Transform {
            children [
            ]
          }

          DEF p148_IntegratorB_Container Transform {
            children [
            ]
          }
        ]
      }

      DEF p148_Edges Group {
        children [
          DEF p148_Negative1_To_p148_FactorA_Container Transform {
            children [
            ]
```

Figure 7-5. Containing transforms

```
            }

            DEF p148_Product_To_p148_IntegrandA_Container Transform {
              children [
              ]
            }

            DEF p148_IntegralA_To_p148_FactorB_Container Transform {
              children [
              ]
            }

            DEF p148_IntegralA_To_p148_IntegrandB_Container Transform {
              children [
              ]
            }

            DEF p148_IntegralB_To_p148_X_Container Transform {
              children [
              ]
            }
          ]
        }
      ]
    }
  }
…
```

Figure 7-5. Continued

Model Hierarchy and Geometry

With this now well defined structure, setting up the presentation file to mimic the model definition hierarchy is simple. Recall that model prototypes were created so that models could be instanced easily. The grouping structure that has been set up is now ready to include these references. For all sub-model components, simply add the appropriate prototype reference to the logical structures. This is how model hierarchy is achieved using this development methodology. Since the example file is defined in one level, there is only one model prototype that includes these references. For more complex models, several prototypes would reference each other in a more complex hierarchical structure.

At this point all that is left is to include the actual scene geometry. It is at this level that Includes should be used since we are at the atomic level of the model definition.

There should be some metaphorical structure for each model prototype included under its container transform. Remember to keep in mind the schema when selecting a metaphorical representation. For the example I simply include an empty VRML *Shape* node for these atomic structures. For a real presentation file, these *Shape* nodes would further include either geometry and appearance nodes, or they would be replaced by a VRML *Include* node. If for some reason, the developer needs access to attributes of these atomic components, *EXTERNPROTO* nodes may be used. Figure 7-6 contains the final structure for the root-model prototype. Appendix B contains the complete example file.

```
…
PROTO p148 [] {
  DEF p148_Handles Group {
    children [
      DEF p148_Outputs Group {
        children [
          DEF p148_X_Container Transform {
            children [
              DEF p148_X Shape {
              }
            ]
          }
        ]
      }
    ]
  }
  DEF p148_Model Group {
    children [
      DEF p148_Nodes Group {
        children [
          DEF p148_Constant_Container Transform {
            children [
              DEF p148_Constant Constant {}
            ]
          }

          DEF p148_Multiplier_Container Transform {
            children [
              DEF p148_Multiplier Multiplier {}
            ]
          }

          DEF p148_IntegratorA_Container Transform {
            children [
              DEF p148_IntegratorA Integrator {}
            ]
          }
```

Figure 7-6. Prototype instances and atomic structures

```
          DEF p148_IntegratorB_Container Transform {
            children [
              DEF p148_IntegratorB Integrator {}
            ]
          }
        ]
      }

      DEF p148_Edges Group {
        children [
          DEF p148_Negative1_To_p148_FactorA_Container Transform {
            children [
              DEF p148_Negative1_To_p148_FactorA Shape {
              }
            ]
          }

          DEF p148_Product_To_p148_IntegrandA_Container Transform {
            children [
              DEF p148_Product_To_p148_IntegrandA Shape {
              }
            ]
          }

          DEF p148_IntegralA_To_p148_FactorB_Container Transform {
            children [
              DEF p148_IntegralA_To_p148_FactorB Shape {
              }
            ]
          }

          DEF p148_IntegralA_To_p148_IntegrandB_Container Transform {
            children [
              DEF p148_IntegralA_To_p148_IntegrandB Shape {
              }
            ]
          }

          DEF p148_IntegralB_To_p148_X_Container Transform {
            children [
              DEF p148_IntegralB_To_p148_X Shape {
              }
            ]
          }
        ]
      }
    ]
  }
}
…
```

Figure 7-6. Continued

CHAPTER 8
COMMENT PARSER

Description

This model was implemented as an example for Fishwick's CAP 5805 Computer Simulation course. The initial idea for this example model came from an existing example for Finite State Machines [7]. The basic idea is to implement a simple scanner that recognizes comment regions in portions of C-style code. This example would be useful as part of a C compiler for recognizing areas of the code that can be ignored by the compiler. Pattern matching happens to be one popular practical application for FSMs and as such I thought that this would make a good example.

In the C family of code languages (C, C++, Java, and so forth) there are two ways to comment code. The first comment-style uses the beginning and ending demarcations "`/*`" and "`*/`", respectively. This allows a programmer to comment regions that span several lines. In this example, this style is referred to as the "C-style" comment. The other comment-style uses only a beginning demarcation, "`//`", the ending being implicitly defined by the End-of-Line (`[EOL]`) character. This allows a programmer to begin a comment anywhere in a line that runs to the end of the line. In this example, this style is referred to as the "C++-style" comment.

Structure

The basic structure for the comment-recognizing FSM is a five-state FSM with 12 transitions as illustrated in Figure 8-1.

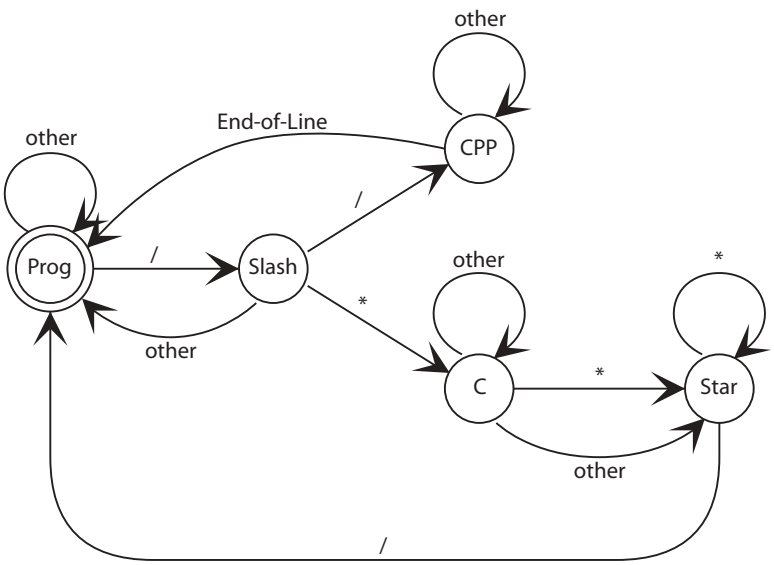

Figure 8-1. Comment scanner FSM

As illustrated, the FSM fully defines *how* to recognize the comment regions, but to actually simulate this model we need to augment the FSM. By definition a scanner takes as input a character stream and outputs a token stream. All that our model is capable of at this point is actually moving from state-space to state-space given an input stream, which we've not yet provided. To solve this problem, we nest the FSM in an FBM that can feed the scanner a comment stream and receive a token stream as output. The final resulting structure, a 3-block, 3-trace FBM with nested FSM, is illustrated in Figure 8-2.

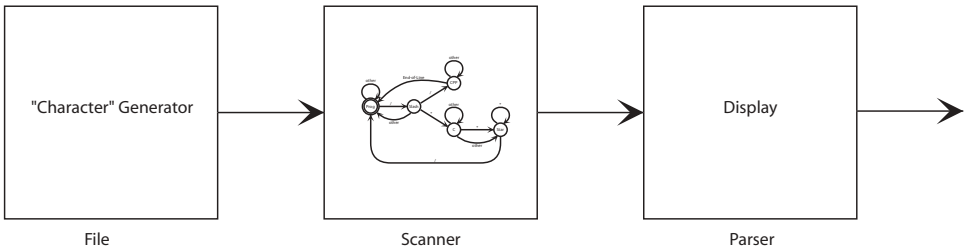

Figure 8-2. Comment scanner FBM

To simulate the model in *RUBE*, we need to do a little planning so that we know how to set up the semantic files. Our first challenge is with the FBM block which feeds characters to the scanner. There are two possibilities for simulating this: providing an

actual file that will be read character-by-character, or simply randomly generating characters. Since we are only interested in recognizing comments—not actually compiling C-code, checking that the code is well-formed, and so forth—we will use a variation of the second approach. In our variation, instead of sending random characters, we will send random numbers that represent random characters. This will make it much easier to tweak the probability distribution of the random number generation to achieve more "interesting" or "realistic" simulations. The generator will generate a random integer between 0 and 4 (inclusive) that will be interpreted as follows:

- 20% chance to generate a "`/`" character, represented by 0
- 20% chance to generate a "`*`" character, represented by 1
- 20% chance to generate a "`[EOL]`" character, represented by 2
- 40% chance to generate a lowercase alphabetical character, represented by 3 and 4

These numbers were chosen for their simplicity and for producing an "interesting" simulation. They are by no means indicative of an analysis of C-style programming to reflect actual "code" to "comment" region proportions in the simulation; this is, after all, only an example.

The second FBM block represents the scanner; this block is driven by the FSM. The FSM transitions are summarized in the Table 8-1 which is interpreted as follows: receipt of the character indicated by the column header while in the state indicated by the row header results in a transition to the state indicated by the cell that is the intersection of the corresponding column and row. This diagram is appropriately referred to as a "transition table" since it maps all possible transitions between states on any given input to the system. For example, if a '/' character is received while in the "program" state, we transition to the "slash" state.

Table 8-1. Scanner transitions summarized

	`/`	`*`	`[EOL]`	`[a-z]`
program	slash	program	program	program
slash	cpp	c	program	program
cpp	cpp	cpp	program	cpp
c	c	star	c	c
star	program	star	c	c

We will need to set up a transition function for each unique transition (the transition table concurs with our diagram that there are twelve of these for our FSM). The transition functions will be relatively simple. They need only check the character received and set the output to "`true`" or "`false`" to indicate whether the transition fires or not. *RUBE* ensures that only possible transitions—transitions *out* of the current state—are analyzed, which is why we need not include this check ourselves.

Similarly, we will need to set up state output; this is the output for the entire FSM. Here again we have two choices regarding our implementation. We can either output simply the current state name, or we can use the states to set flags or other special variables. Since it is the job of the scanner to output a token stream, according to our model semantics we would ideally use the second method and set a special variable indicating whether we are in a comment region or not. However, at the time of this example's implementation this was not possible and so we used the first method instead.[1] This means that the state functions are simple as well, merely setting the output to the name of the current state.

[1] This is now possible in *RUBE* and is reflected in the following case-study in Chapter 9.

The third FBM block receives the scanner's output for further processing. In this example, no further processing is necessary, so we simply use this block as a display mechanism.[2] For a model representing an entire compiler, this block, would be the Parser—hence the block's name—which would take as input a token stream (as generated from the scanner) and output a parse tree.

The entire code for model structure and semantics—MXL and JavaScript—is available in Appendix C.

Mapping

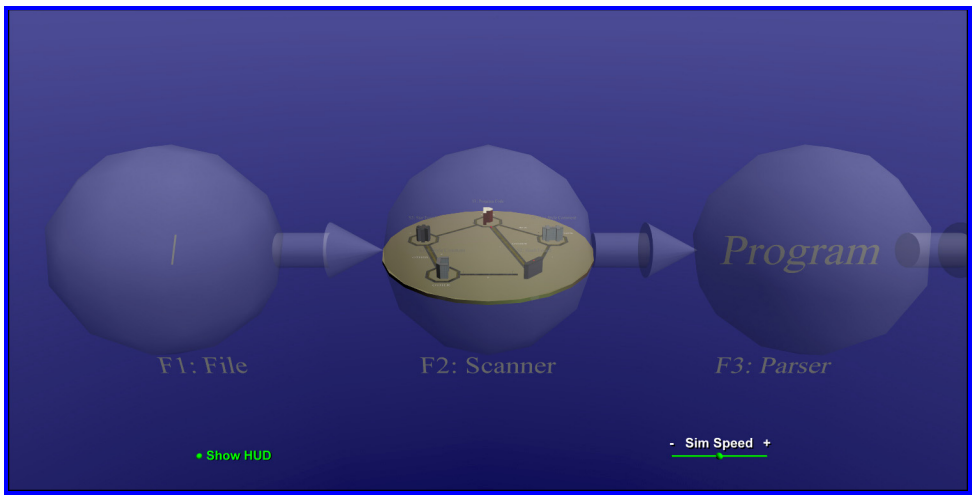

Figure 8-3. Comment scanner virtual world (112 kB VRML file)
http://www.cise.ufl.edu/~kdamkjer/rube/worlds/comment/comment.wrl

The FBM model uses extremely general metaphors in this example only because it is of little interest to us and thus we want to diminish the visual impact it has on our model world. It is integral to the model functionality, however, so the possibility of omitting it completely from the model representation was rejected. The model that we're really interested in is the FSM nested within the second block of the FBM, which stands

[2] For this implementation we check the state name received and display the code or comment region appropriately. If we had used the second method, this would merely echo the input received from the second FBM block, which would be ideal given the model semantics.

out prominently against the translucent FBM. The FBM uses metaphor of model as an environment with the overly-general metaphor of nodes as containers, as illustrated in Figure 8-3. The traces between models are represented as arrows indicating the directionality of the flow of information through the model system. You may notice that model handles are not directly represented in this model world. This is an artifact from the stage our research was at when this model was developed. At that point, the model interface elements had not yet been introduced into MXL and we had not yet realized metaphorical representations for them. The closest representation that they have in this model is the "threshold" metaphor. Yet even though boundaries are clearly defined between model components, portals are not, so even this metaphor is somewhat lacking in this example.

Even though the FBM is of little interest to us, its presence in the model world is in fact very useful. By having the blocks present, we are able to display block output as a form of model feedback during simulation. Each of the blocks in the model displays its output: Block 1 displays the character "read" from the file, Block 2 displays the current Scanner state, and Block 3 displays the interpretation of the Scanner state as being indicative of a comment or program region of code. Figures 8-4–8-6 show each of the blocks responding to a "/" character being read in. Block 1 simply echoes the character read. Block 2 indicates the current state by lighting a sphere at the top of one of the buildings. Figure 8-5 is not the best on this indicator, Figures 8-8 and 8-12 offer much better perspectives of the red sphere lit atop the "slash" building. Block 3 correctly interprets that the "slash" state is indicative of being in a program region.

The Scanner FSM uses architectural metaphors for its representations. Each of the states in the FSM is represented by a building. To help the user distinguish between the buildings quickly and easily, the shape of the building was chosen to reflect the state it represents. Thus the “program” state is shaped like the letter “P”, the “slash” state is

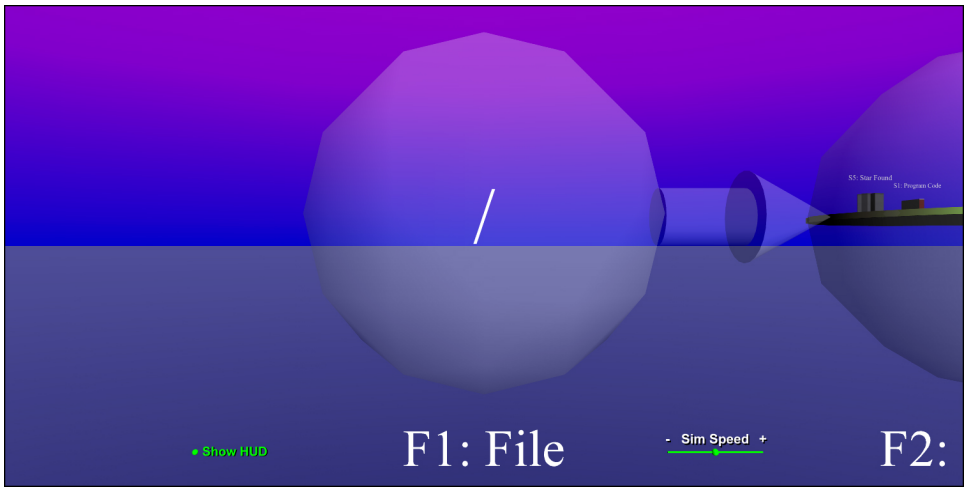

Figure 8-4. Block: File

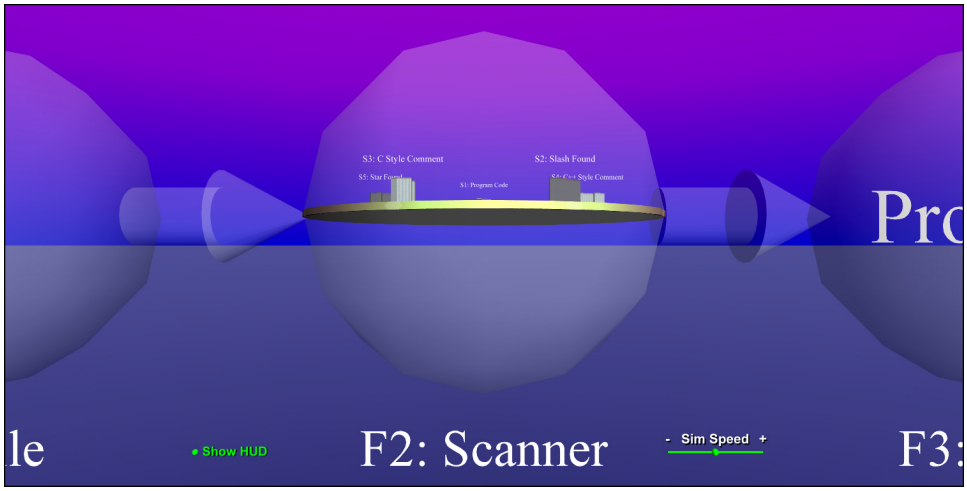

Figure 8-5. Block: Scanner

Figure 8-6. Block: Parser

shaped like the forward-slash character, the C++-style state is shaped like two adjoining "plus" signs, the C-style state is shaped like the letter "C" and the "star" state is shaped like an asterisk. Figures 8-7–8-11 offer close-up perspectives of each of the buildings.

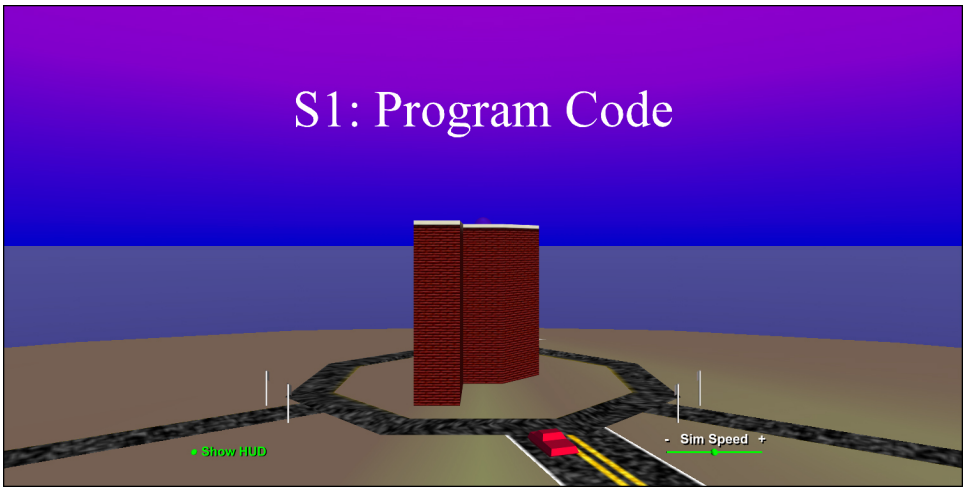

Figure 8-7. State: Program

Figure 8-8. State: Slash

Transitions are also represented using the "conduit" metaphor that works well with the "city" schema employed by the Scanner FSM: roadways. The directionality of the transitions are indicated by either a divided highway—which actually represents *two* transitions, one in each direction—or a one-way street as indicated by appropriate "one-way" signs at the heads of the transitions and "wrong-way" signs at the tails. The

Figure 8-9. State: CPP

Figure 8-10. State: C

Figure 8-11. State: Star

resulting transitions are illustrated in Figure 8-12. To observe the "one-way" and "wrong-way" signs you may need to refer to Figures 8-7–8-11 or 8-13. Even if you can't read the signs, the transition directionality can still be discerned by noticing that "wrong-way" signs come in pairs while "one-way" signs appear alone.

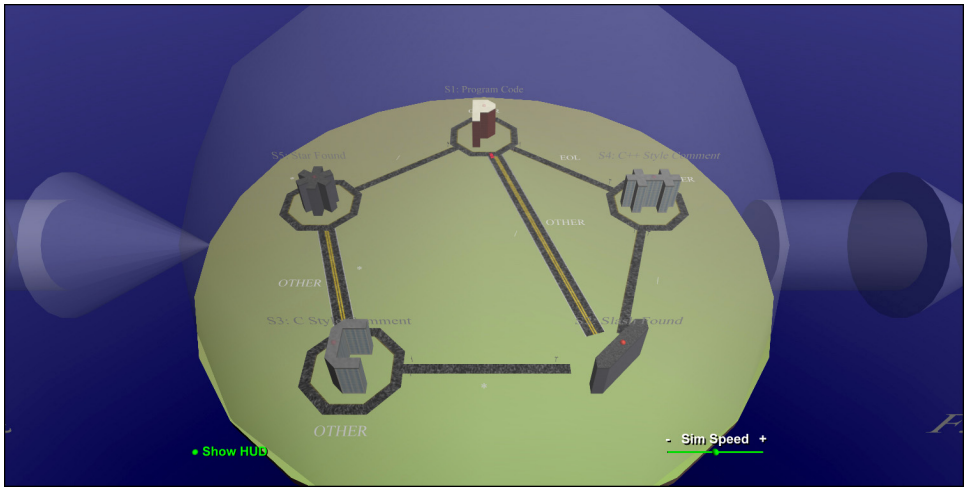

Figure 8-12. Transitions

Figure 8-13. Signage indicating transition directionality

CHAPTER 9
BARNSLEY FERN IFS

Description and Definition

This example was chosen primarily for its complexity and ability to surface multi-modeling concerns in *RUBE* and MXL. The model, at least its most popular output presentation, is also fairly familiar to several people outside the modeling and simulation community, and thus it is hoped that this example would provide more insight into not only the aesthetics of the model output, but of the model content as well and that it would be exemplary for demonstrating aesthetic approaches to modeling and simulation. The model that I am referring to is the Barnsley Fern, one member of a family of fractal geometries called Iterated Function Systems (IFS).

The IFS, in general, consists of a set of contractive affine transformations that are applied recursively to a piece of geometry to produce an attractor image. The attractor for the Barnsley Fern is, as the name suggests, a very convincing fern represented by a set of only four such transformations. This approach to generating fractal geometry is often referred to as a Multiple-Reduction Copy Machine (MRCM), the idea being that the attractors generated by the IFS, at least in two dimensions, are the same as if you repeatedly made copies of the copies produced by a copy machine with several lenses that each reduced and transformed the original in some way.

Though this metaphor is excellent for visualizing the function of the set of affine transformations, it should be noted that several members of the IFS family of fractal geometries do not tend to their attractors quickly when using this method. The Barnsley

Fern happens to be one of these "problem children". To correct for this situation, the fact that applying any one of the set of affine transformations to any point in the attractor generates another point in the attractor is used to increase the speed with which the attractor is generated. By selecting one of the transforms at random and applying it to the previously generated point, we will eventually generate the attractor to an acceptable degree of precision.[1] When using this approach, it should be observed that the probability distribution used to determine the selected of the transformations need not be uniform, and further, that ideal results are seldom achieved using uniform distributions. In general, the transformations which reduce the original *less* should have a *higher* probability of being selected since they are responsible for generating larger portions of the attractor image. This is where the advantage of this approach is seen since the relatively larger areas of the attractor that were problematic before can be made to generate at the same rate as relatively smaller areas with correctly selected distributions.

To generate the new point in the attractor set, a transformation matrix is applied to the previously generated point. This transformation matrix encapsulates all of the affine transformations for one of the transformation sets in the IFS and it is calculated through a series of matrix multiplications. In $\mathbb{R}^3$, there are twelve rudimentary transform matrices that are combined into the single affine transformation matrix.[2] Nine of these twelve—the three transformations representing scale and the six representing shear—combine

[1] To "perfectly" generate the attractor, we would have to allow this and any other fractal algorithm to run infinitely. Such is the nature of fractal geometry: infinitely self-similar.

[2] Most of the time, these values are pre-computed to optimize performance. We leave the calculation in the model definition and simulation since we are interested more in demonstrating the complete process than in quickly generating the attractor image.

nicely into a single matrix for the transform group by simple placing elements in the correct position in the matrix as outlined in Table 9-1.

Table 9-1. Affine transformations in $\mathbb{R}^3$

Transform	Matrix Name	Transform Matrix
Scale	Sc	$\begin{bmatrix} Sc_x & 0 & 0 \\ 0 & Sc_y & 0 \\ 0 & 0 & Sc_z \end{bmatrix}$
Shear	Sh	$\begin{bmatrix} 1 & Sh_{xy} & Sh_{xz} \\ Sh_{yx} & 1 & Sh_{yz} \\ Sh_{zx} & Sh_{zy} & 1 \end{bmatrix}$
Rotation about the *x*-axis	Ro_x	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_x & \sin\theta_x \\ 0 & -\sin\theta_x & \cos\theta_x \end{bmatrix}$
Rotation about the *y*-axis	Ro_y	$\begin{bmatrix} \cos\theta_y & 0 & -\sin\theta_y \\ 0 & 1 & 0 \\ \sin\theta_y & 0 & \cos\theta_y \end{bmatrix}$
Rotation about the *z*-axis	Ro_z	$\begin{bmatrix} \cos\theta_z & \sin\theta_z & 0 \\ -\sin\theta_z & \cos\theta_z & 0 \\ 0 & 0 & 1 \end{bmatrix}$

These matrices are then applied to each other to produce the final affine transformation matrix as follows:[3]

Let $L = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$ represent a transformation in $\mathbb{R}^3$.

Then $L = (Sc)(Sh)(Ro_x(Ro_y(Ro_z)))$.

Observe that L now represents all transformations in $\mathbb{R}^3$ except translation. Translations "augment" the affine transformation matrix as follows:

[3] The order of application of the rotation matrices *does* matter. I've chosen a *xyz* rotation order.

Let $T = \begin{bmatrix} j \\ k \\ l \end{bmatrix} = \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix}$ be a translation in $\mathbb{R}^3$.

Then A is the augmented affine transformation matrix defined as follows:

$$A = (L \mid T) = \left[\begin{array}{ccc|c} a & b & c & j \\ d & e & f & k \\ g & h & i & l \end{array}\right]$$

To generate a new point in the attractor, simply apply this augmented matrix to the previously generated point as follows:

$$x_{n+1} = ax_n + by_n + cz_n + j$$

$$y_{n+1} = dx_n + ey_n + fz_n + k$$

$$z_{n+1} = gx_n + hy_n + iz_n + l$$

Structure

The complete multi-model representation for the Barnsley Fern IFS is quite complex. I will take a top down approach to explaining the multi-model. Complete MXL and JavaScript model definition is available in Appendix D.

IFS FBM

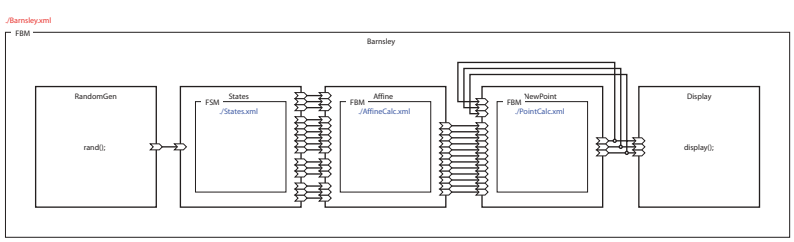

Figure 9-1. The general IFS FBM

At the top, most abstract, level, the IFS is defined as a five block FBM. The first block is a random number generator that generates a floating-point number between 0

and 1. The random numbers are generated with a uniform distribution. The second block takes as input the random number generated by Block 1; this number is used to select a set of transformations (scales, shears, rotations, and translations) based on an inherent probability distribution. The third block combines the transformations from the second block, without translations, into a single affine transformation matrix. Block 4 applies this matrix—augmented with the translation values from the second block—to the previously generated point to generate a new point in the attractor. The fifth block displays the cumulative output of the system: the attractor image.

Transformation Selection FSM

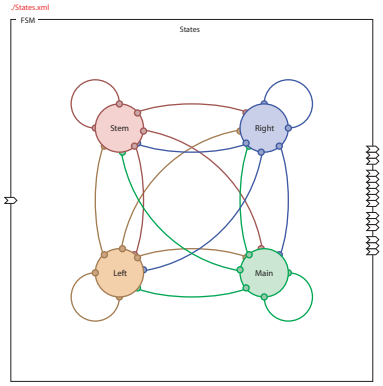

Figure 9-2. Transformation states FSM

Within the transformation selection block, there is a specialized FSM that uses probabilities to drive its transitions called a Markov Model. While *RUBE* does not yet support this model natively, it is easy enough to implement using the random number generator and nested FSM.[4] The nested FSM for the Barnsley Fern has four states (one for each of the possible transforms) that are fully connected, meaning we have sixteen transitions (including self-referencing cycles). This structure is derived from the structure for the general IFS in which there is a state for each transformation set and each

[4] With this model type natively supported, there would be no need for the first functional block.

state is reachable from every state (including itself). Each state, when active, sets a complete set of variables representing the rudimentary transformations in three-dimensions: rotation about the *x*, *y* and *z* axes; scale along the *x*, *y*, and *z* axes; shear *x* along *y*, *x* along *z*, *y* along *x*, *y* along *z*, *z* along *x*, and *z* along *y*; and translate along *x*, *y*, and *z* axes.

Augmented Affine Transformation Matrix Calculation FBM

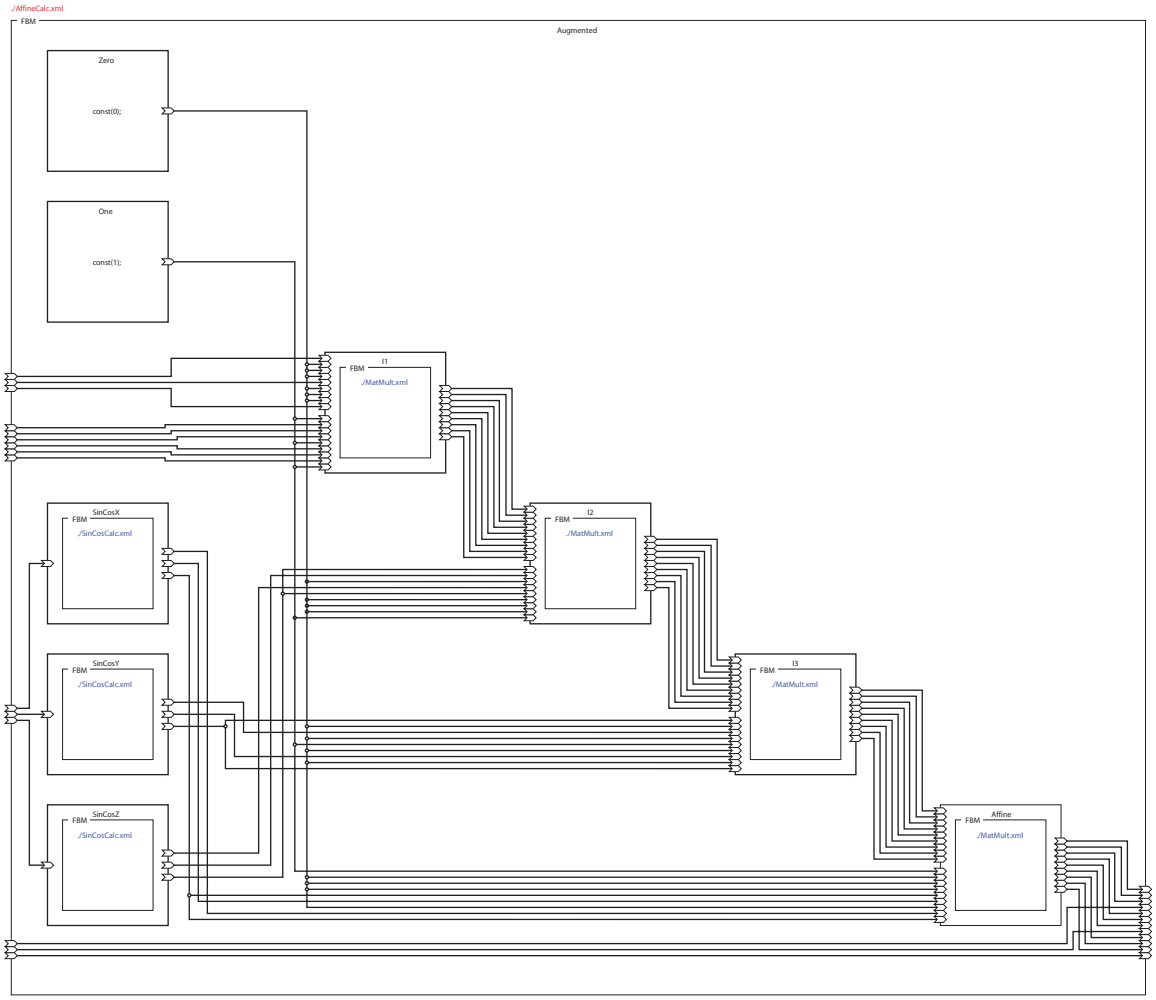

Figure 9-3. Augmented affine transformation matrix calculation

Each of these transformations is combined into the single augmented affine transformation matrix in this FBM. In the FBM, we must modify some of the raw transform data before it can be used in the series of matrix multiplications (four blocks).

Specifically, the rotation values must be converted to their respective sine, cosine and negative sine values, which are then fed to the matrix multiplication blocks (three blocks). The final two blocks in this model are constant number generators, which generate a constant one or zero for use in various matrices as illustrated in Table 9-1.

Sine, Cosine, and Negative Sine Calculation FBM

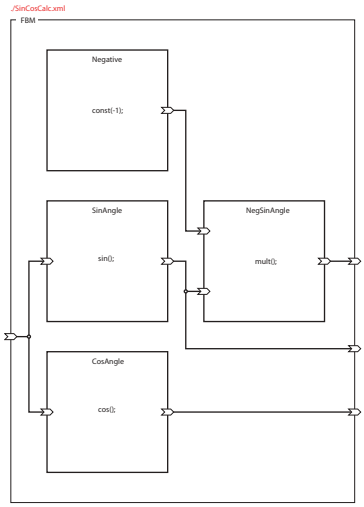

Figure 9-4. Sine, cosine, and negative sine calculation FBM

This FBM simply takes as input an angle and uses script blocks to calculate the sine and cosine of the angle (two blocks). The negative sine is calculated by simply multiplying the sine by negative one (two blocks).

Three-By-Three By Three-By-Three Matrix Multiplication FBM

Matrix multiplication can be viewed as a combination of vector multiplications. For a three-by-three by three-by-three matrix multiplication there are nine such multiplications each of which are represented by a block in this model as illustrated in Figure 9-5.

Row-Column Product (Vector Multiplication) FBM

Row-column product calculation is essentially just a vector multiplication, which is the sum of the products of corresponding elements in each of the vectors. The blocks in

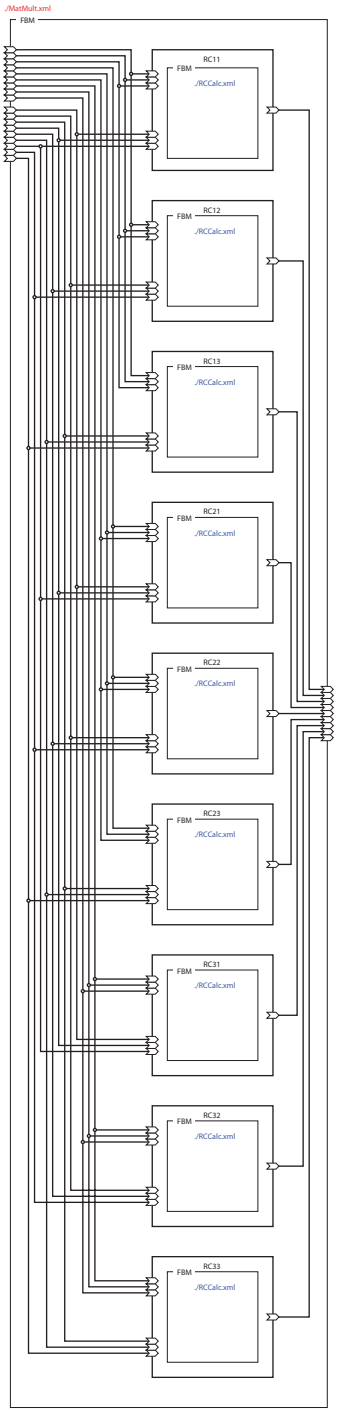

Figure 9-5. Matrix multiplication FBM

this model, illustrated in Figure 9-6, multiply the corresponding elements together, and then sum the products to produce the vector product.

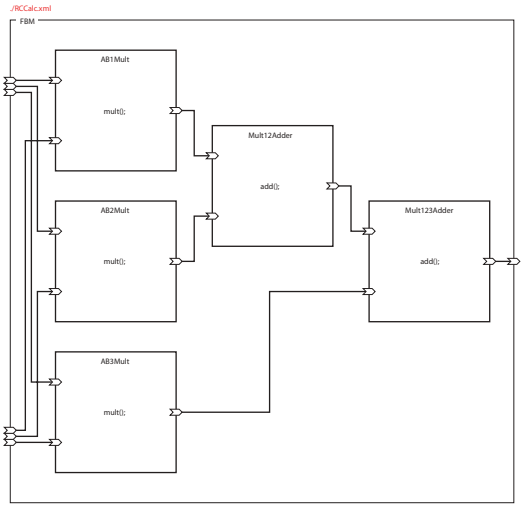

Figure 9-6. Row-column product calculation FBM

New Point Calculation FBM

The new point calculation model is simply a block representation of the equations used to calculate new points in the attractor set. The model treats the application of the affine matrix as a series of vector products and then adds translation values to the resulting products to get the final coordinates for the new point.

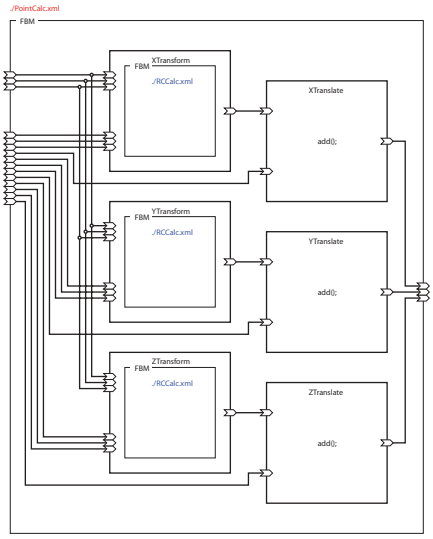

Figure 9-7. New point calculation FBM

Mapping

The IFS FBM model representation uses the metaphor of model is a building, nodes are machines. The model in its entirety is represented as a conservatory, or green house. Each of the blocks within the model is represented by a specialized machine: the random number generator is represented as a fountain; the transform selection is represented as a water-driven virtual analog device, the augmented affine transformation matrix calculator is represented as an air-pressure–driven virtual analog device, the new-point calculator is represented by an air-pressure–driven virtual three-dimensional "printer", and the display block is represented by a virtual sculpture of the IFS attractor.

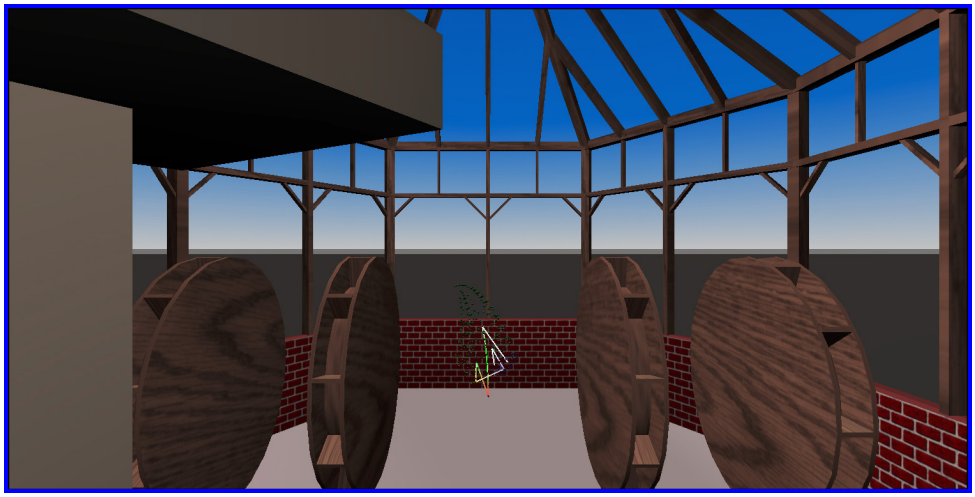

Figure 9-8. Barnsley Conservatory (10 kB VRML file)
http://www.cise.ufl.edu/~kdamkjer/rube/worlds/Barnsley/rube.wrl

The water-driven virtual analog device that represents the transformation selection FSM uses the model is a machine and blocks are machine parts metaphor. In this representation each of the states is represented by a water wheel that turns at a unique rate. The wheels drive hammers which set the output air pressure.

The augmented affine transformation matrix calculation FBM and matrix multiplication FBM act largely as routing mechanisms since there is no actual calculation that takes place within these models. All calculation is done by sub-models. As such, these models use the metaphor of model is a machine with nodes are machine parts. The air-pressures fed to these models are in turn fed to sub-models to drive smaller virtual analog devices which perform the actual calculations.

One of these rudimentary sub-models is the sine and cosine calculation FBM. This FBM uses the metaphor of model is a machine and nodes are machine parts. In this model, edges are represented by interconnection between the machine parts, that is, that the nodes are directly influencing each other; this makes sense at such a rudimentary model level. The sine and cosine machine must take air pressure as its input and return air pressure as its output to fit with the parent schema. This works out quite well. This model representation uses incoming air pressure, representing the rotation angle, to drive pistons which set air pressure reflecting the sine and cosine of the incoming value. The negative sine that is to be calculated by this model needs to be a negative pressure equal in magnitude to the sine, this is exactly what we achieve by placing a piston arm adjacent to the sine piston, and placing the attachment point to the driving gear on the opposite side, representing a multiplication of -1. This will make the two pistons complements of one another. Note that the same approach could be used for cosine; we just don't have a need for that functionality in this model.

The other rudimentary model is the vector multiplication calculation FBM. In this model the air-pressures are used to drive compressors/vacuums which represent the

multiplication blocks. The resulting pressures are then summed via a network of valves—representing addition—to produce the single vector product value.

The air-pressures calculated by the network of virtual analog devices representing the augmented affine transformation matrix calculator are then passed to the new point calculation FBM. Again, we use the metaphor of model is a machine and nodes are machine parts. The air-pressures that feed into the device control the positioning of two lasers which produce a point in the attractor at their point of intersection.

This virtual three-dimensional sculpture, representing the display block, is the end result of the complex network of virtual analog devices.

CHAPTER 10
CONCLUSION

Summary of Results and Future Work

The goals of this research were to refine the Multi-model Exchange Language to remove ambiguities in model definitions and to facilitate language extensions, to extend the *RUBE* framework to allow for model definition and semantic component reuse, to explore model definition development to take advantage of the multi-model paradigm, to explore metaphorical mappings for model representations, and to establish a methodology for creating presentation worlds for models developed with the *RUBE* framework.

For the first goal of refining MXL, this research was successful in establishing a basic structure for the MXL language and in formalizing generic model structure and model cross communication mechanisms. The work that remains to be done on the schema—which may simply be an ongoing process—is in implementing new model types. As Appendix A shows, there is a basic skeletal structure for a new type of function model—the System-Dynamics Model—and a new type of declarative model—the Petri Network—in MXL. These model types need to be further refined and tested. Also new model families should be added. The most pressing need seems to be for equation-based models, queuing models, and constraint models.

The second objective of extending *RUBE* to take advantage of model definition and semantic component reuse was successful for the model types we currently have implemented. For any added model type, developers will need to be sure to test that the

MXL massive merge still performs as expected. The proper use of the model interface elements should ensure this, but we have yet to test adding a new model type to the existing MXL structure.

The exploration of model development strategies and metaphorical mappings for model representations was instrumental in the later development of the methodology for creating presentation worlds. The study of metaphorical mappings for model representations is so new that I feel that my research has only barely scratched the surface of what may be possible. I feel that this has definite potential for future research. The list of metaphors that I provided was quite long, but it is most likely just a cursory glance at the metaphors that are embedded in our profession and in the field of modeling and simulation.

Auxiliary Objectives

There were a few auxiliary objectives that I was asked to consider while performing my research. These objectives were basically to formalize my thoughts on possibly partially automating the model representation development process and extending the established methodology to representations other than VRML.

Automation

The possibility for partially automating the model development process is very good. There is a high correlation between the MXL model definition and the resulting VRML scene graph hierarchy. The biggest challenge that I can foresee in this area would be extracting unique node definitions from the MXL file, possibly by comparing the *src* attributes on the model elements, for conversion into model prototypes. The prototype concept is not a bad one and may be of use in MXL, though that will be for future research to determine.

Extension to Representations other than VRML

As for extending the methodology to other representations, I again see the prototype definition as the limiting factor. Most 3-D and 2-D development packages have grouping or parenting structures that are similar to the VRML *Transform* and *Group* nodes, however, so the hierarchical extension should not be a problem. To implement the prototype functionality, I would explore these packages referencing capabilities. The only problem that I've found is that all too often these tend to operate as VRML's *Include* node, where you lose all access to the included model's sub-attributes, or they simply place a complete copy of the referenced file in the scene graph, which makes for very large complex scenes without a nice interface for cross-communication. In short, I believe the potential exists, though there may need to be more research done on how to implement prototype functionality in the other representations.

APPENDIX A
MXL SCHEMA DEFINITION

```
<?xml version="1.0" encoding="utf-8"?>
<xsd:schema targetNamespace="http://www.cise.ufl.edu/~kdamkjer/rube/MXL"
xmlns="http://www.cise.ufl.edu/~kdamkjer/rube/MXL" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
elementFormDefault="unqualified" attributeFormDefault="unqualified">
  <xsd:annotation>
    <xsd:documentation>Multi-Model Exchange Language (MXL) schema definition.  Copyright 2003
University of Florida.  All rights reserved.</xsd:documentation>
  </xsd:annotation>
  <xsd:simpleType name="fieldTypeName">
    <xsd:annotation>
      <xsd:appinfo>All supported data types should be placed in this list.  Future data types should be
appended to the end of the list.</xsd:appinfo>
      <xsd:documentation>Enumerates all MXL data types.</xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Boolean">
        <xsd:annotation>
          <xsd:appinfo>Boolean is analogous to xsd:boolean type.</xsd:appinfo>
          <xsd:documentation>Boolean is used to represent a logical type with possible values (true | false)
to match the XML boolean type.</xsd:documentation>
        </xsd:annotation>
      </xsd:enumeration>
      <xsd:enumeration value="Booleans">
        <xsd:annotation>
          <xsd:appinfo>Booleans is analogous to xsd:boolean list type.</xsd:appinfo>
          <xsd:documentation>Booleans is used to represent an array of Boolean
values.</xsd:documentation>
        </xsd:annotation>
      </xsd:enumeration>
      <xsd:enumeration value="Integer">
        <xsd:annotation>
          <xsd:appinfo>Integer is used analogous to xsd:integer type.</xsd:appinfo>
          <xsd:documentation>Integer is used to represent a 32-bit integer type.</xsd:documentation>
        </xsd:annotation>
      </xsd:enumeration>
      <xsd:enumeration value="Integers">
        <xsd:annotation>
          <xsd:appinfo>Integers is used analogous to xsd:integer list type.</xsd:appinfo>
          <xsd:documentation>Integers is used to represent an array of Integer
values.</xsd:documentation>
        </xsd:annotation>
      </xsd:enumeration>
      <xsd:enumeration value="Float">
        <xsd:annotation>
          <xsd:appinfo>Float is used analogous to xsd:float type.</xsd:appinfo>
          <xsd:documentation>Float is used to represent a single-precision floating-point
type.</xsd:documentation>
        </xsd:annotation>
      </xsd:enumeration>
      <xsd:enumeration value="Floats">
```

Figure A-1. MXL schema definition

```
            <xsd:annotation>
              <xsd:appinfo>Floats is used analogous to xsd:float list type.</xsd:appinfo>
              <xsd:documentation>Floats is used to represent an array of Float values.</xsd:documentation>
            </xsd:annotation>
          <xsd:enumeration value="Floats">
            <xsd:annotation>
              <xsd:appinfo>Floats is used analogous to xsd:float list type.</xsd:appinfo>
              <xsd:documentation>Floats is used to represent an array of Float values.</xsd:documentation>
            </xsd:annotation>
          </xsd:enumeration>
          <xsd:enumeration value="Double">
            <xsd:annotation>
              <xsd:appinfo>Double is analogous to xsd:double type.</xsd:appinfo>
              <xsd:documentation>Double is used to represent a double-precision floating-point
type.</xsd:documentation>
            </xsd:annotation>
          </xsd:enumeration>
          <xsd:enumeration value="Doubles">
            <xsd:annotation>
              <xsd:appinfo>Doubles is analogous to xsd:double list type.</xsd:appinfo>
              <xsd:documentation>Doubles is used to represent an array of Double
values.</xsd:documentation>
            </xsd:annotation>
          </xsd:enumeration>
          <xsd:enumeration value="String">
            <xsd:annotation>
              <xsd:appinfo>String is used analogous to an xsd:string type.</xsd:appinfo>
              <xsd:documentation>String is used to represent a character string type.</xsd:documentation>
            </xsd:annotation>
          </xsd:enumeration>
          <xsd:enumeration value="Strings">
            <xsd:annotation>
              <xsd:appinfo>Strings is used analogous to an xsd:string list type.</xsd:appinfo>
              <xsd:documentation>Strings is used to represent an array of String values.</xsd:documentation>
            </xsd:annotation>
          </xsd:enumeration>
        </xsd:restriction>
      </xsd:simpleType>
      <xsd:element name="MXL">
        <xsd:annotation>
          <xsd:appinfo>The MXL file is broken into two main sections: Model Definition and Simulation
Frame.</xsd:appinfo>
          <xsd:documentation>This is the MXL root node.</xsd:documentation>
        </xsd:annotation>
        <xsd:complexType>
          <xsd:sequence>
            <xsd:group ref="models"/>
            <xsd:element name="simulation" minOccurs="0">
              <xsd:complexType>
                <xsd:attribute name="start_time" type="xsd:float" use="required"/>
                <xsd:attribute name="end_time" type="xsd:float" use="required"/>
                <xsd:attribute name="delta_time" type="xsd:float" use="required"/>
                <xsd:attribute name="cycle_time" type="xsd:float" use="required"/>
              </xsd:complexType>
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:group name="models">
        <xsd:annotation>
```

Figure A-1. Continued

```
        <xsd:appinfo>All model categories should be placed under the "choice" element.  New model
categories should be appended to the end of the current list.</xsd:appinfo>
        <xsd:documentation>Generic Model Group</xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
        <xsd:group ref="handles"/>
        <xsd:choice>
          <xsd:group ref="functionalModels"/>
          <xsd:group ref="declarativeModels"/>
        </xsd:choice>
      </xsd:sequence>
    </xsd:group>
    <xsd:group name="handles">
      <xsd:annotation>
        <xsd:documentation>Handle Group</xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
        <xsd:element name="input" minOccurs="0" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:attribute name="id" type="xsd:ID" use="required"/>
            <xsd:attribute name="datatype" type="fieldTypeName" use="required"/>
            <xsd:attribute name="index" type="xsd:unsignedInt" use="required"/>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="output" minOccurs="0" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:attribute name="id" type="xsd:ID" use="required"/>
            <xsd:attribute name="datatype" type="fieldTypeName" use="required"/>
            <xsd:attribute name="initial" type="xsd:NMTOKEN" use="optional"/>
            <xsd:attribute name="index" type="xsd:unsignedInt" use="required"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:group>
    <xsd:group name="functionalModels">
      <xsd:annotation>
        <xsd:documentation>Functional Model Group</xsd:documentation>
      </xsd:annotation>
      <xsd:choice>
        <xsd:group ref="scripts"/>
        <xsd:group ref="fbm"/>
        <xsd:group ref="sdm"/>
      </xsd:choice>
    </xsd:group>
    <xsd:group name="scripts">
      <xsd:annotation>
        <xsd:documentation>Script Group</xsd:documentation>
      </xsd:annotation>
      <xsd:choice>
        <xsd:element name="script">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="param" minOccurs="0" maxOccurs="unbounded">
                <xsd:complexType>
                  <xsd:attribute name="value" type="xsd:NMTOKEN" use="required"/>
                  <xsd:attribute name="index" type="xsd:unsignedInt" use="required"/>
                  <!--<xsd:attribute name="name" type="xsd:token" use="required"/>-->
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
            <xsd:attribute name="lang" use="required">
```

Figure A-1. Continued

```
                <xsd:simpleType>
                  <xsd:restriction base="xsd:string">
                    <xsd:enumeration value="JavaScript"/>
                    <xsd:enumeration value="Java"/>
                  </xsd:restriction>
                </xsd:simpleType>
              </xsd:attribute>
              <xsd:attribute name="src" type="xsd:anyURI" use="required"/>
              <xsd:attribute name="func" type="xsd:NCName" use="required"/>
            </xsd:complexType>
          </xsd:element>
        </xsd:choice>
      </xsd:group>
      <xsd:group name="fbm">
        <xsd:sequence>
          <xsd:element name="fbm">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:group ref="fbmNodes" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:group ref="fbmEdges" minOccurs="0" maxOccurs="unbounded"/>
              </xsd:sequence>
              <xsd:attribute name="id" type="xsd:ID" use="required"/>
              <xsd:attribute name="src" type="xsd:anyURI" use="optional"/>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:group>
      <xsd:group name="fbmNodes">
        <xsd:choice>
          <xsd:element name="block">
            <xsd:complexType>
              <xsd:choice>
                <xsd:group ref="models"/>
              </xsd:choice>
              <xsd:attribute name="id" type="xsd:ID" use="required"/>
            </xsd:complexType>
          </xsd:element>
        </xsd:choice>
      </xsd:group>
      <xsd:group name="fbmEdges">
        <xsd:choice>
          <xsd:element name="trace" minOccurs="0">
            <xsd:complexType>
              <xsd:attribute name="from" type="xsd:IDREF" use="required"/>
              <xsd:attribute name="to" type="xsd:IDREF" use="required"/>
            </xsd:complexType>
          </xsd:element>
        </xsd:choice>
      </xsd:group>
      <xsd:group name="sdm">
        <xsd:sequence>
          <xsd:element name="sdm">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:group ref="sdmNodes" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:group ref="sdmEdges" minOccurs="0" maxOccurs="unbounded"/>
              </xsd:sequence>
              <xsd:attribute name="id" type="xsd:ID" use="required"/>
              <xsd:attribute name="src" type="xsd:anyURI" use="optional"/>
            </xsd:complexType>
          </xsd:element>
```

Figure A-1. Continued

```
    </xsd:sequence>
  </xsd:group>
  <xsd:group name="sdmNodes">
    <xsd:choice>
      <xsd:element name="src">
        <xsd:complexType>
          <xsd:attribute name="id" type="xsd:ID" use="required"/>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="sink">
        <xsd:complexType>
          <xsd:attribute name="id" type="xsd:ID" use="required"/>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="rate">
        <xsd:complexType>
          <xsd:choice>
            <xsd:group ref="models"/>
          </xsd:choice>
          <xsd:attribute name="id" type="xsd:ID" use="required"/>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="level">
        <xsd:complexType>
          <xsd:choice>
            <xsd:group ref="models"/>
          </xsd:choice>
          <xsd:attribute name="id" type="xsd:ID" use="required"/>
          <xsd:attribute name="initial" type="xsd:double" use="required"/>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="aux">
        <xsd:complexType>
          <xsd:choice>
            <xsd:group ref="models"/>
          </xsd:choice>
          <xsd:attribute name="id" type="xsd:ID" use="required"/>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="const">
        <xsd:complexType>
          <xsd:attribute name="id" type="xsd:ID" use="required"/>
          <xsd:attribute name="value" type="xsd:double" use="required"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:choice>
  </xsd:group>
  <xsd:group name="sdmEdges">
    <xsd:choice>
      <xsd:element name="arc">
        <xsd:complexType>
          <xsd:attribute name="type" use="required">
            <xsd:simpleType>
              <xsd:restriction base="xsd:string">
                <xsd:enumeration value="flow"/>
                <xsd:enumeration value="cause-effect"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:attribute>
          <xsd:attribute name="from" type="xsd:IDREF" use="required"/>
          <xsd:attribute name="to" type="xsd:IDREF" use="required"/>
```

Figure A-1. Continued

```
            </xsd:complexType>
        </xsd:element>
    </xsd:choice>
</xsd:group>
<xsd:group name="declarativeModels">
    <xsd:annotation>
        <xsd:documentation>Declarative Model Group</xsd:documentation>
    </xsd:annotation>
    <xsd:choice>
        <xsd:group ref="fsm"/>
        <xsd:group ref="p-net"/>
    </xsd:choice>
</xsd:group>
<xsd:group name="fsm">
    <xsd:all>
        <xsd:element name="fsm">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:group ref="fsmNodes" minOccurs="0" maxOccurs="unbounded"/>
                    <xsd:group ref="fsmEdges" minOccurs="0" maxOccurs="unbounded"/>
                </xsd:sequence>
                <xsd:attribute name="id" type="xsd:ID" use="required"/>
                <xsd:attribute name="src" type="xsd:anyURI" use="optional"/>
            </xsd:complexType>
        </xsd:element>
    </xsd:all>
</xsd:group>
<xsd:group name="fsmNodes">
    <xsd:choice>
        <xsd:element name="state">
            <xsd:complexType>
                <xsd:choice>
                    <xsd:group ref="models"/>
                </xsd:choice>
                <xsd:attribute name="id" type="xsd:ID" use="required"/>
                <xsd:attribute name="start" type="xsd:boolean" use="optional" default="false"/>
            </xsd:complexType>
        </xsd:element>
    </xsd:choice>
</xsd:group>
<xsd:group name="fsmEdges">
    <xsd:choice>
        <xsd:element name="transition">
            <xsd:complexType>
                <xsd:choice>
                    <xsd:group ref="models"/>
                </xsd:choice>
                <xsd:attribute name="from" type="xsd:IDREF" use="required"/>
                <xsd:attribute name="to" type="xsd:IDREF" use="required"/>
            </xsd:complexType>
        </xsd:element>
    </xsd:choice>
</xsd:group>
<xsd:group name="p-net">
    <xsd:all>
        <xsd:element name="p-net">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:group ref="p-netNodes" minOccurs="0" maxOccurs="unbounded"/>
                    <xsd:group ref="p-netEdges" minOccurs="0" maxOccurs="unbounded"/>
                </xsd:sequence>
```

Figure A-1. Continued

```
                <xsd:attribute name="id" type="xsd:ID" use="required"/>
                <xsd:attribute name="src" type="xsd:anyURI" use="optional"/>
            </xsd:complexType>
        </xsd:element>
    </xsd:all>
  </xsd:group>
  <xsd:group name="p-netNodes">
    <xsd:choice>
      <xsd:element name="condition">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="token" minOccurs="0" maxOccurs="unbounded">
              <xsd:complexType>
                <xsd:attribute name="color" type="xsd:nonNegativeInteger" use="optional" default="0"/>
                <xsd:attribute name="init_tokens" type="xsd:nonNegativeInteger" use="optional"
default="0"/>
              </xsd:complexType>
            </xsd:element>
            <xsd:choice>
              <xsd:group ref="models" minOccurs="0"/>
            </xsd:choice>
          </xsd:sequence>
          <xsd:attribute name="id" type="xsd:ID" use="required"/>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="event">
        <xsd:complexType>
          <xsd:choice>
            <xsd:group ref="models"/>
          </xsd:choice>
          <xsd:attribute name="id" type="xsd:ID" use="required"/>
          <xsd:attribute name="time" type="xsd:double" use="optional" default="0.0"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:choice>
  </xsd:group>
  <xsd:group name="p-netEdges">
    <xsd:choice>
      <xsd:element name="connection">
        <xsd:complexType>
          <xsd:choice>
            <xsd:group ref="models" minOccurs="0"/>
          </xsd:choice>
          <xsd:attribute name="from" type="xsd:IDREF" use="required"/>
          <xsd:attribute name="to" type="xsd:IDREF" use="required"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:choice>
  </xsd:group>
</xsd:schema>
```

Figure A-1. Continued

APPENDIX B
MODEL PRESENTATION EXAMPLE STRUCTURE

```
#VRML V2.0 utf8

PROTO Constant [] {
  DEF Constant_Handles Group {
    children [
      DEF Constant_Outputs Group {
        children [
          DEF Constant_Value_Container Transform {
            children [
              DEF Constant_Value Shape {
              }
            ]
          }
        ]
      }
    ]
  }

  DEF Constant_Model Group {
    children [
      DEF Constant_Nodes Group {
        children [
          DEF Constant_Container Transform {
            children [
              DEF Constant Shape {
              }
            ]
          }
        ]
      }
    ]
  }
}

PROTO Multiplier [] {
  DEF Multiplier_Handles Group {
    children [
      DEF Multiplier_Inputs Group {
        children [
          DEF Multiplier_Factor1_Container Transform {
            children [
              DEF Multiplier_Factor1 Shape {
              }
            ]
          }
          DEF Multiplier_Factor2_Container Transform {
            children [
              DEF Multiplier_Factor2 Shape {
              }
            ]
          }
```

Figure B-1. Model presentation example file

```
        ]
      }

      DEF Multiplier_Outputs Group {
        children [
          DEF Multiplier_Product_Container Transform {
            children [
              DEF Multiplier_Product Shape {
              }
            ]
          }
        ]
      }
    ]
  }

  DEF Multiplier_Model Group {
    children [
      DEF Multiplier_Nodes Group {
        children [
          DEF Multiplier_Container Transform {
            children [
              DEF Multiplier Shape {
              }
            ]
          }
        ]
      }
    ]
  }
}

PROTO Integrator [] {
  DEF Integrator_Handles Group {
    children [
      DEF Integrator_Inputs Group {
        children [
          DEF Integrator_Integrand_Container Transform {
            children [
              DEF Integrator_Integrand Shape {
              }
            ]
          }
        ]
      }

      DEF Integrator_Outputs Group {
        children [
          DEF Integrator_Integral_Container Transform {
            children [
              DEF Integrator_Integral Shape {
              }
            ]
          }
        ]
      }
    ]
  }

  DEF Integrator_Model Group {
    children [
      DEF Integrator_Nodes Group {
```

Figure B-1. Continued

```
        children [
          DEF Integrator_Container Transform {
            children [
              DEF Integrator Shape {
              }
            ]
          }
        ]
      }
    ]
  }
}

PROTO p148 [] {
  DEF p148_Handles Group {
    children [
      DEF p148_Outputs Group {
        children [
          DEF p148_X_Container Transform {
            children [
              DEF p148_X Shape {
              }
            ]
          }
        ]
      }
    ]
  }

  DEF p148_Model Group {
    children [
      DEF p148_Nodes Group {
        children [
          DEF p148_Constant_Container Transform {
            children [
              DEF p148_Constant Constant {}
            ]
          }

          DEF p148_Multiplier_Container Transform {
            children [
              DEF p148_Multiplier Multiplier {}
            ]
          }

          DEF p148_IntegratorA_Container Transform {
            children [
              DEF p148_IntegratorA Integrator {}
            ]
          }
          DEF p148_IntegratorB_Container Transform {
            children [
              DEF p148_IntegratorB Integrator {}
            ]
          }
        ]
      }

      DEF p148_Edges Group {
        children [
          DEF p148_Negative1_To_p148_FactorA_Container Transform {
            children [
```

Figure B-1. Continued

```
                DEF p148_Negative1_To_p148_FactorA Shape {
                }
              ]
            }

            DEF p148_Product_To_p148_IntegrandA_Container Transform {
              children [
                DEF p148_Product_To_p148_IntegrandA Shape {
                }
              ]
            }

            DEF p148_IntegralA_To_p148_FactorB_Container Transform {
              children [
                DEF p148_IntegralA_To_p148_FactorB Shape {
                }
              ]
            }

            DEF p148_IntegralA_To_p148_IntegrandB_Container Transform {
              children [
                DEF p148_IntegralA_To_p148_IntegrandB Shape {
                }
              ]
            }

            DEF p148_IntegralB_To_p148_X_Container Transform {
              children [
                DEF p148_IntegralB_To_p148_X Shape {
                }
              ]
            }
          ]
        }
      ]
    }
}

DEF p148 p148 {}
```

Figure B-1. Continued

APPENDIX C
MODEL DEFINITION FOR THE C-STYLE COMMENT SCANNER

Comment Scanner FBM: comment.xml

```
<?xml version="1.0" encoding="utf-8"?>

<mxl:MXL xmlns:mxl="http://www.cise.ufl.edu/~kdamkjer/rube/MXL"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.cise.ufl.edu/~kdamkjer/rube/MXL
http://www.cise.ufl.edu/~kdamkjer/rube/MXL/MXL.xsd">

   <output id="InComment" datatype="Boolean" index="0"/>

   <fbm id="Comment">

      <!-- State: Slash -->
      <block id="File">
         <output id="Character" datatype="Integer" index="0"/>
         <script lang="JavaScript" src="comment.js" func="input"/>
      </block>

      <!-- State: Slash -->
      <block id="Scanner">
         <input id="ReadIn" datatype="Integer" index="0"/>
         <output id="TokenState" datatype="String" index="0"/>
         <fsm id="Scanner_FSM" src="comment_sub.xml"/>
      </block>

      <!-- State: Slash -->
      <block id="Parser">
         <input id="Scanner_State" datatype="String" index="0"/>
         <output id="Comment_Region" datatype="Boolean" index="0"/>
         <script lang="JavaScript" src="comment.js" func="handle"/>
      </block>

      <trace from="Character" to="ReadIn"/>
      <trace from="TokenState" to="Scanner_State"/>
      <trace from="Comment_Region" to="InComment"/>

   </fbm>

   <simulation start_time="0" end_time="1000" delta_time="1" cycle_time="4"/>

</mxl:MXL>
```

Figure C-1. comment.xml

Comment Scanner FSM: comment_sub.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<mxl:MXL xmlns:mxl="http://www.cise.ufl.edu/~kdamkjer/rube/MXL"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.cise.ufl.edu/~kdamkjer/rube/MXL
http://www.cise.ufl.edu/~kdamkjer/rube/MXL/MXL.xsd">

   <input id="Trigger" datatype="Integer" index="0"/>

   <output id="State" datatype="String" index="0"/>

   <fsm id="Match_FSM">

      <!-- State: Program -->

      <state id="Program" start="true">
         <script lang="JavaScript" src="comment.js" func="program"/>
      </state>

      <!-- State: Slash -->
      <state id="Slash">
         <script lang="JavaScript" src="comment.js" func="slash"/>
      </state>

      <!-- State: C -->
      <state id="C">
         <script lang="JavaScript" src="comment.js" func="cStyle"/>
      </state>

      <!-- State: CPP -->
      <state id="CPP">
         <script lang="JavaScript" src="comment.js" func="cppStyle"/>
      </state>

      <!-- State: Program -->
      <state id="Star">
         <script lang="JavaScript" src="comment.js" func="star"/>
      </state>

      <!-- Transitions out of Program -->
      <transition from="Program" to="Slash">
         <script lang="JavaScript" src="comment.js" func="programToSlash"/>
      </transition>
      <transition from="Program" to="Program">
         <script lang="JavaScript" src="comment.js" func="programToProgram"/>
      </transition>

      <!-- Transitions out of Slash -->
      <transition from="Slash" to="C">
         <script lang="JavaScript" src="comment.js" func="slashToC"/>
      </transition>
      <transition from="Slash" to="CPP">
         <script lang="JavaScript" src="comment.js" func="slashToCPP"/>
      </transition>
      <transition from="Slash" to="Program">
         <script lang="JavaScript" src="comment.js" func="slashToProgram"/>
      </transition>
```

Figure C-2. comment_sub.xml

```
    <!-- Transitions out of C -->
    <transition from="C" to="Star">
      <script lang="JavaScript" src="comment.js" func="cToStar"/>
    </transition>
    <transition from="C" to="C">
      <script lang="JavaScript" src="comment.js" func="cToC"/>
    </transition>

    <!-- Transitions out of CPP -->
    <transition from="CPP" to="Program">
      <script lang="JavaScript" src="comment.js" func="cppToProgram"/>
    </transition>
    <transition from="CPP" to="CPP">
      <script lang="JavaScript" src="comment.js" func="cppToCPP"/>
    </transition>

    <!-- Transitions out of Star -->
    <transition from="Star" to="Star">
      <script lang="JavaScript" src="comment.js" func="starToStar"/>
    </transition>
    <transition from="Star" to="Program">
      <script lang="JavaScript" src="comment.js" func="starToProgram"/>
    </transition>
    <transition from="Star" to="C">
      <script lang="JavaScript" src="comment.js" func="starToC"/>
    </transition>
  </fsm>
</mxl:MXL>
```

Figure C-2. Continued

Comment Scanner JavaScript: comment.js

```
/*************************************************************************\
 * FBM functions                                                         *
 *-----------------------------------------------------------------------*
 * F1: gen()                                                             *
 * F2: N/A                                                               *
 * F3: disp()                                                            *
\*************************************************************************/

/*************************************************************************\
 * function: gen()                                                       *
 *-----------------------------------------------------------------------*
 * Generates a random "character" from the our alphabet as follows:      *
 *   0 - '/'                                                             *
 *   1 - '*'                                                             *
 *   2 - [EOL]                                                           *
 *   3 - [a-z]                                                           *
 *   4 - [a-z]                                                           *
\*************************************************************************/

function gen() {
  this.output[0] = Math.floor(Math.random()*5);
}

/*************************************************************************\
 * function: disp()                                                      *
```

Figure C-3. comment.js

```
 *-------------------------------------------------------------------------*
 * Display function, shows whether the model is in a comment or code      *
 *   region.                                                               *
\***************************************************************************/

function disp() {
  if (this.input[0] == "Program" || this.input[0] == "Slash") {
    this.output[0] = false;
  }
  else {
    this.output[0] = true;
  }
}

//=========================================================================\\

/***************************************************************************\
 * FSM functions                                                           *
 *-------------------------------------------------------------------------*
 * S1   - "Prog"        : program()                                        *
 * S2   - "Slash"       : slash()                                          *
 * S3   - "C"           : cStyle()                                         *
 * S4   - "CPP"         : cppStyle()                                       *
 * S5   - "Star"        : star()                                           *
 *-------------------------------------------------------------------------*
 * TR11 - "Other_01"    : progToProg()                                     *
 * TR12 - "Slash_01"    : progToSlash()                                    *
 * TR21 - "Other_02"    : slashToProg()                                    *
 * TR23 - "Star_01"     : slashToC()                                       *
 * TR24 - "Slash_02"    : slashToCPP()                                     *
 * TR33 - "Other_03"    : cToC()                                           *
 * TR35 - "Star_02"     : cToStar()                                        *
 * TR44 - "Other_04"    : cppToCPP()                                       *
 * TR41 - "EOL_01"      : cppToProg()                                      *
 * TR53 - "Other_05"    : starToC()                                        *
 * TR55 - "Star_03"     : starToStar()                                     *
 * TR51 - "Slash_03"    : starToProg()                                     *
\***************************************************************************/

/***************************************************************************\
 * function: prog()                                                        *
 *-------------------------------------------------------------------------*
 * Returns "Program" when S1 - "Program" is the active state               *
\***************************************************************************/

function program() {
  this.output[0] = "Program";
}

/***************************************************************************\
 * function: slash()                                                       *
 *-------------------------------------------------------------------------*
 * Returns "Slash" when S2 - "Slash" is the active state                   *
\***************************************************************************/

function slash() {
  this.output[0] = "Slash";
}
```

Figure C-3. Continued

```
/*************************************************************************\
 * function: cStyle()                                                    *
 *-----------------------------------------------------------------------*
 * Returns "C" when S3 - "C" is the active state                         *
\*************************************************************************/

function cStyle() {
  this.output[0] = "C";
}

/*************************************************************************\
 * function: cppStyle()                                                  *
 *-----------------------------------------------------------------------*
 * Returns "CPP" when S4 - "CPP" is the active state                     *
\*************************************************************************/

function cppStyle() {
  this.output[0] = "CPP";
}

/*************************************************************************\
 * function: star()                                                      *
 *-----------------------------------------------------------------------*
 * Returns "Star" when S5 - "Star" is the active state                   *
\*************************************************************************/

function star() {
  this.output[0] = "Star";
}

//-----------------------------------------------------------------------\\

/*************************************************************************\
 * function: programToProgram()                                          *
 *-----------------------------------------------------------------------*
 * Sets TR11 to fire when any token other than 0 - "/" is received       *
\*************************************************************************/

function programToProgram() {
  if(parseInt(this.input[0]) != 0)
    this.output[0] = true;
  else
    this.output[0] = false;
}

/*************************************************************************\
 * function: programToSlash()                                            *
 *-----------------------------------------------------------------------*
 * Sets TR12 to fire when token 0 - "/" is received                      *
\*************************************************************************/

function programToSlash() {
  if(parseInt(this.input[0]) == 0)
    this.output[0] = true;
  else
    this.output[0] = false;
}

/*************************************************************************\
```

Figure C-3. Continued

```
 * function: slashToProgram()                                          *
 *-------------------------------------------------------------------*
 * Sets TR21 to fire when any token other than 0 - "/" or 1 - "*" is  *
 *   received                                                          *
\*********************************************************************/

function slashToProgram() {
  if(parseInt(this.input[0]) >= 2)
    this.output[0] = true;
  else
    this.output[0] = false;
}

/*********************************************************************\
 * function: slashToC()                                                *
 *-------------------------------------------------------------------*
 * Sets TR23 to fire when token 1 - "*" is received                    *
\*********************************************************************/

function slashToC() {
  if(parseInt(this.input[0]) == 1)
    this.output[0] = true;
  else
    this.output[0] = false;
}

/*********************************************************************\
 * function: slashToCPP()                                              *
 *-------------------------------------------------------------------*
 * Sets TR24 to fire when token 0 - "/" is received                    *
\*********************************************************************/

function slashToCPP() {
  if(parseInt(this.input[0]) == 0)
    this.output[0] = true;
  else
    this.output[0] = false;
}

/*********************************************************************\
 * function: cToC()                                                    *
 *-------------------------------------------------------------------*
 * Sets TR33 to fire when any token other than 1 - "*" is received     *
\*********************************************************************/

function cToC() {
  if(parseInt(this.input[0]) != 1)
    this.output[0] = true;
  else
    this.output[0] = false;
}

/*********************************************************************\
 * function: cToStar()                                                 *
 *-------------------------------------------------------------------*
 * Sets TR35 to fire when token 1 - "*" is received                    *
\*********************************************************************/

function cToStar() {
```

Figure C-3. Continued

```
  if(parseInt(this.input[0]) == 1)
    this.output[0] = true;
  else
    this.output[0] = false;
}

/*************************************************************************\
 * function: cppToCPP()                                                  *
 *-----------------------------------------------------------------------*
 * Sets TR44 to fire when any token other than 2 - "EOL" is received     *
\*************************************************************************/

function cppToCPP() {
  if(parseInt(this.input[0]) != 2)
    this.output[0] = true;
  else
    this.output[0] = false;
}

/*************************************************************************\
 * function: cppToProgram()                                              *
 *-----------------------------------------------------------------------*
 * Sets TR41 to fire when token 2 - "EOL" is received                    *
\*************************************************************************/

function cppToProgram() {
  if(parseInt(this.input[0]) == 2)
    this.output[0] = true;
  else
    this.output[0] = false;
}

/*************************************************************************\
 * function: starToC()                                                   *
 *-----------------------------------------------------------------------*
 * Sets TR53 to fire when any token other than 0 - "/" or 1 - "*" is     *
 *   received                                                            *
\*************************************************************************/

function starToC() {
  if(parseInt(this.input[0]) >= 2)
    this.output[0] = true;
  else
    this.output[0] = false;
}

/*************************************************************************\
 * function: starToStar()                                                *
 *-----------------------------------------------------------------------*
 * Sets TR55 to fire when token 1 - "*" is received                      *
\*************************************************************************/

function starToStar() {
  if(parseInt(this.input[0]) == 1)
    this.output[0] = true;
  else
    this.output[0] = false;
}
```

Figure C-3. Continued

```
/*************************************************************************\
 * function: starToProgram()                                             *
 *-----------------------------------------------------------------------*
 * Sets TR51 to fire when token 0 - "/" is received                      *
\*************************************************************************/

function starToProgram() {
  if(parseInt(this.input[0]) == 0)
    this.output[0] = true;
  else
    this.output[0] = false;
}
```

Figure C-3. Continued

APPENDIX D
MODEL DEFINITION FOR THE BARNSLEY FERN IFS

Barnsley Fern IFS: Barnsley.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<mxl:MXL xmlns:mxl="http://www.cise.ufl.edu/~kdamkjer/rube/MXL"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.cise.ufl.edu/~kdamkjer/rube/MXL
http://www.cise.ufl.edu/~kdamkjer/rube/MXL/MXL.xsd">
   <fbm id="Barnsley">

      <!-- Random Number Generator -->
      <block id="RandomGen">
         <output id="Random" datatype="Double" index="0"/>
         <script lang="JavaScript" src="./math.js" func="rand"/>
      </block>

      <!-- Stochastic Transform Selection -->
      <block id="States">

         <!-- Input Trigger -->
         <input id="Trigger" datatype="Double" index="0"/>

         <!-- Scale Outputs -->
         <output id="ScX" datatype="Double" index="0"/>
         <output id="ScY" datatype="Double" index="1"/>
         <output id="ScZ" datatype="Double" index="2"/>

         <!-- Shear Outputs -->
         <output id="ShXY" datatype="Double" index="3"/>
         <output id="ShXZ" datatype="Double" index="4"/>
         <output id="ShYX" datatype="Double" index="5"/>
         <output id="ShYZ" datatype="Double" index="6"/>
         <output id="ShZX" datatype="Double" index="7"/>
         <output id="ShZY" datatype="Double" index="8"/>

         <!-- Rotation Outputs -->
         <output id="RX" datatype="Double" index="9"/>
         <output id="RY" datatype="Double" index="10"/>
         <output id="RZ" datatype="Double" index="11"/>

         <!-- Translation Outputs -->
         <output id="TX" datatype="Double" index="12"/>
         <output id="TY" datatype="Double" index="13"/>
         <output id="TZ" datatype="Double" index="14"/>
         <fsm id="Fronds" src="./States.xml"/>
      </block>

      <!-- Affine Transformation Matrix Calculator -->
      <block id="Affine">
```

Figure D-1. Barnsley.xml

```
    <!-- Scale Inputs -->
    <input id="Scale_X" datatype="Double" index="0"/>
    <input id="Scale_Y" datatype="Double" index="1"/>
    <input id="Scale_Z" datatype="Double" index="2"/>

    <!-- Shear Inputs -->
    <input id="Shear_XY" datatype="Double" index="3"/>
    <input id="Shear_XZ" datatype="Double" index="4"/>
    <input id="Shear_YX" datatype="Double" index="5"/>
    <input id="Shear_YZ" datatype="Double" index="6"/>
    <input id="Shear_ZX" datatype="Double" index="7"/>
    <input id="Shear_ZY" datatype="Double" index="8"/>

    <!-- Rotation Inputs -->
    <input id="Rotate_X" datatype="Double" index="9"/>
    <input id="Rotate_Y" datatype="Double" index="10"/>
    <input id="Rotate_Z" datatype="Double" index="11"/>

    <!-- Translation Inputs -->
    <input id="Translate_X" datatype="Double" index="12"/>
    <input id="Translate_Y" datatype="Double" index="13"/>
    <input id="Translate_Z" datatype="Double" index="14"/>

    <!-- Affine Transformation Matrix Output -->
    <output id="Augmented11" datatype="Double" index="0"/>
    <output id="Augmented12" datatype="Double" index="1"/>
    <output id="Augmented13" datatype="Double" index="2"/>
    <output id="Augmented14" datatype="Double" index="3"/>
    <output id="Augmented21" datatype="Double" index="4"/>
    <output id="Augmented22" datatype="Double" index="5"/>
    <output id="Augmented23" datatype="Double" index="6"/>
    <output id="Augmented24" datatype="Double" index="7"/>
    <output id="Augmented31" datatype="Double" index="8"/>
    <output id="Augmented32" datatype="Double" index="9"/>
    <output id="Augmented33" datatype="Double" index="10"/>
    <output id="Augmented34" datatype="Double" index="11"/>
    <fbm id="AffineCalc" src="./AffineCalc.xml"/>
  </block>

  <!-- New Point Calculator -->
  <block id="NewPoint">

    <!-- Current Point -->
    <input id="CurrentX" datatype="Double" index="0" initial="0.5"/>
    <input id="CurrentY" datatype="Double" index="1" initial="0.5"/>
    <input id="CurrentZ" datatype="Double" index="2" initial="0.5"/>

    <!-- Augmented Affine Matrix -->
    <input id="Transform11" datatype="Double" index="3"/>
    <input id="Transform12" datatype="Double" index="4"/>
    <input id="Transform13" datatype="Double" index="5"/>
    <input id="Transform14" datatype="Double" index="6"/>
    <input id="Transform21" datatype="Double" index="7"/>
    <input id="Transform22" datatype="Double" index="8"/>
    <input id="Transform23" datatype="Double" index="9"/>
    <input id="Transform24" datatype="Double" index="10"/>
    <input id="Transform31" datatype="Double" index="11"/>
    <input id="Transform32" datatype="Double" index="12"/>
    <input id="Transform33" datatype="Double" index="13"/>
    <input id="Transform34" datatype="Double" index="14"/>
```

Figure D-1. Continued

```
        <!-- New Point -->
        <output id="NewX" datatype="Double" index="0"/>
        <output id="NewY" datatype="Double" index="1"/>
        <output id="NewZ" datatype="Double" index="2"/>
        <fbm id="PointCalc" src="PointCalc.xml"/>
      </block>

      <!-- Display -->
      <block id="Display">
        <input id="DispX" datatype="Double" index="0"/>
        <input id="DispY" datatype="Double" index="1"/>
        <input id="DispZ" datatype="Double" index="2"/>
        <script lang="JavaScript" src="./Barnsley" func="display"/>
      </block>

      <!-- Random Number to the Stochastic Section -->
      <trace from="Random" to="Trigger"/>

      <!-- Transformation Information to the Affine Calculator -->
      <trace from="ScX" to="Scale_X"/>
      <trace from="ScY" to="Scale_Y"/>
      <trace from="ScZ" to="Scale_Z"/>
      <trace from="ShXY" to="Shear_XY"/>
      <trace from="ShXZ" to="Shear_XZ"/>
      <trace from="ShYX" to="Shear_YX"/>
      <trace from="ShYZ" to="Shear_YZ"/>
      <trace from="ShZX" to="Shear_ZX"/>
      <trace from="ShZY" to="Shear_ZY"/>
      <trace from="RX" to="Rotate_X"/>
      <trace from="RY" to="Rotate_Y"/>
      <trace from="RZ" to="Rotate_Z"/>
      <trace from="TX" to="Translate_X"/>
      <trace from="TY" to="Translate_Y"/>
      <trace from="TZ" to="Translate_Z"/>

      <!-- Affine Matrix to the New Point Calculator -->
      <trace from="Augmented11" to="Transform11"/>
      <trace from="Augmented12" to="Transform12"/>
      <trace from="Augmented13" to="Transform13"/>
      <trace from="Augmented14" to="Transform14"/>
      <trace from="Augmented21" to="Transform21"/>
      <trace from="Augmented22" to="Transform22"/>
      <trace from="Augmented23" to="Transform23"/>
      <trace from="Augmented24" to="Transform24"/>
      <trace from="Augmented31" to="Transform31"/>
      <trace from="Augmented32" to="Transform32"/>
      <trace from="Augmented33" to="Transform33"/>
      <trace from="Augmented34" to="Transform34"/>

      <!-- New Point becomes Current Point -->
      <trace from="NewX" to="CurrentX"/>
      <trace from="NewY" to="CurrentY"/>
      <trace from="NewZ" to="CurrentZ"/>

      <!-- Display New Point -->
      <trace from="NewX" to="DispX"/>
      <trace from="NewY" to="DispY"/>
      <trace from="NewZ" to="DispZ"/>
    </fbm>
  </mxl:MXL>
```

Figure D-1. Continued

IFS States: States.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<mxl:MXL xmlns:mxl="http://www.cise.ufl.edu/~kdamkjer/rube/MXL"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.cise.ufl.edu/~kdamkjer/rube/MXL
http://www.cise.ufl.edu/~kdamkjer/rube/MXL/MXL.xsd">

  <!-- Input Trigger -->
  <input id="Random" datatype="Double" index="0"/>

  <!-- Scale Outputs -->
  <output id="ScX" datatype="Double" index="0"/>
  <output id="ScY" datatype="Double" index="1"/>
  <output id="ScZ" datatype="Double" index="2"/>

  <!-- Shear Outputs -->
  <output id="ShXY" datatype="Double" index="3"/>
  <output id="ShXZ" datatype="Double" index="4"/>
  <output id="ShYX" datatype="Double" index="5"/>
  <output id="ShYZ" datatype="Double" index="6"/>
  <output id="ShZX" datatype="Double" index="7"/>
  <output id="ShZY" datatype="Double" index="8"/>

  <!-- Rotation Outputs -->
  <output id="RX" datatype="Double" index="9"/>
  <output id="RY" datatype="Double" index="10"/>
  <output id="RZ" datatype="Double" index="11"/>

  <!-- Translation Outputs -->
  <output id="TX" datatype="Double" index="12"/>
  <output id="TY" datatype="Double" index="13"/>
  <output id="TZ" datatype="Double" index="14"/>
  <fsm id="States">

    <!-- States -->
    <state id="Stem" start="true">

      <!-- Stem Transformations -->
      <script lang="JavaScript" src="./Barnsley.js" func="stemGen"/>
    </state>
    <state id="LeftFrond">

      <!-- Left Transformations -->
      <script lang="JavaScript" src="./Barnsley.js" func="leftGen"/>
    </state>
    <state id="RightFrond">

      <!-- Right Transformations -->
      <script lang="JavaScript" src="./Barnsley.js" func="rightGen"/>
    </state>
    <state id="MainFrond">

      <!-- Main Transformations -->
      <script lang="JavaScript" src="./Barnsley.js" func="mainGen"/>
    </state>

    <!-- Stem Transitions -->
    <transition from="Stem" to="Stem">
      <script lang="JavaScript" src="./Barnsley.js" func="toStem"/>
    </transition>
```

Figure D-2. States.xml

```
      <transition from="Stem" to="LeftFrond">
        <script lang="JavaScript" src="./Barnsley.js" func="toLeftFrond"/>
      </transition>
      <transition from="Stem" to="RightFrond">
        <script lang="JavaScript" src="./Barnsley.js" func="toRightFrond"/>
      </transition>
      <transition from="Stem" to="MainFrond">
        <script lang="JavaScript" src="./Barnsley.js" func="toMainFrond"/>
      </transition>

      <!-- Left Frond Transitions -->
      <transition from="LeftFrond" to="Stem">
        <script lang="JavaScript" src="./Barnsley.js" func="toStem"/>
      </transition>
      <transition from="LeftFrond" to="LeftFrond">
        <script lang="JavaScript" src="./Barnsley.js" func="toLeftFrond"/>
      </transition>
      <transition from="LeftFrond" to="RightFrond">
        <script lang="JavaScript" src="./Barnsley.js" func="toRightFrond"/>
      </transition>
      <transition from="LeftFrond" to="MainFrond">
        <script lang="JavaScript" src="./Barnsley.js" func="toMainFrond"/>
      </transition>

      <!-- Right Frond Transitions -->
      <transition from="RightFrond" to="Stem">
        <script lang="JavaScript" src="./Barnsley.js" func="toStem"/>
      </transition>
      <transition from="RightFrond" to="LeftFrond">
        <script lang="JavaScript" src="./Barnsley.js" func="toLeftFrond"/>
      </transition>
      <transition from="RightFrond" to="RightFrond">
        <script lang="JavaScript" src="./Barnsley.js" func="toRightFrond"/>
      </transition>
      <transition from="RightFrond" to="MainFrond">
        <script lang="JavaScript" src="./Barnsley.js" func="toMainFrond"/>
      </transition>

      <!-- Main Frond Transitions -->
      <transition from="MainFrond" to="Stem">
        <script lang="JavaScript" src="./Barnsley.js" func="toStem"/>
      </transition>
      <transition from="MainFrond" to="LeftFrond">
        <script lang="JavaScript" src="./Barnsley.js" func="toLeftFrond"/>
      </transition>
      <transition from="MainFrond" to="RightFrond">
        <script lang="JavaScript" src="./Barnsley.js" func="toRightFrond"/>
      </transition>
      <transition from="MainFrond" to="MainFrond">
        <script lang="JavaScript" src="./Barnsley.js" func="toMainFrond"/>
      </transition>
    </fsm>
  </mxl:MXL>
```

Figure D-2. Continued

Affine Transformation Matrix Calculation: AffineCalc.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<mxl:MXL xmlns:mxl="http://www.cise.ufl.edu/~kdamkjer/rube/MXL"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.cise.ufl.edu/~kdamkjer/rube/MXL
http://www.cise.ufl.edu/~kdamkjer/rube/MXL/MXL.xsd">

  <!-- Scale Inputs -->
  <input id="Scale_X" datatype="Double" index="0"/>
  <input id="Scale_Y" datatype="Double" index="1"/>
  <input id="Scale_Z" datatype="Double" index="2"/>

  <!-- Shear Inputs -->
  <input id="Shear_XY" datatype="Double" index="3"/>
  <input id="Shear_XZ" datatype="Double" index="4"/>
  <input id="Shear_YX" datatype="Double" index="5"/>
  <input id="Shear_YZ" datatype="Double" index="6"/>
  <input id="Shear_ZX" datatype="Double" index="7"/>
  <input id="Shear_ZY" datatype="Double" index="8"/>

  <!-- Rotation Inputs -->
  <input id="Rotate_X" datatype="Double" index="9"/>
  <input id="Rotate_Y" datatype="Double" index="10"/>
  <input id="Rotate_Z" datatype="Double" index="11"/>

  <!-- Translation Inputs -->
  <input id="Translate_X" datatype="Double" index="12"/>
  <input id="Translate_Y" datatype="Double" index="13"/>
  <input id="Translate_Z" datatype="Double" index="14"/>

  <!-- Affine Transformation Matrix Output -->
  <output id="Augmented11" datatype="Double" index="0"/>
  <output id="Augmented12" datatype="Double" index="1"/>
  <output id="Augmented13" datatype="Double" index="2"/>
  <output id="Augmented14" datatype="Double" index="3"/>
  <output id="Augmented21" datatype="Double" index="4"/>
  <output id="Augmented22" datatype="Double" index="5"/>
  <output id="Augmented23" datatype="Double" index="6"/>
  <output id="Augmented24" datatype="Double" index="7"/>
  <output id="Augmented31" datatype="Double" index="8"/>
  <output id="Augmented32" datatype="Double" index="9"/>
  <output id="Augmented33" datatype="Double" index="10"/>
  <output id="Augmented34" datatype="Double" index="11"/>

  <!-- Augmented Affine Calculation FBM -->
  <fbm id="Augmented">

    <!-- Constant Zero -->
    <block id="Zero">
      <output id="Const_Zero" datatype="Double" index="0"/>
      <script lang="JavaScript" src="./math.js" func="constant">
        <param index="0" value="0"/>
      </script>
    </block>

    <!-- Constant One -->
    <block id="One">
      <output id="Const_One" datatype="Double" index="0"/>
      <script lang="JavaScript" src="./math.js" func="constant">
        <param index="0" value="1"/>
```

Figure D-3. Affine.xml

```
    </script>
  </block>

  <!-- Sine, Cosine, and Negative Sin of X Rotation Angle -->
  <block id="SinCosX">
    <input id="AngleX" datatype="Double" index="0"/>
    <output id="SinX" datatype="Double" index="0"/>
    <output id="CosX" datatype="Double" index="1"/>
    <output id="NegSinX" datatype="Double" index="2"/>
    <fbm id="XSinCosCalc" src="./SinCosCalc.xml"/>
  </block>

  <!-- Sine, Cosine, and Negative Sin of Y Rotation Angle -->
  <block id="SinCosY">
    <input id="AngleY" datatype="Double" index="0"/>
    <output id="SinY" datatype="Double" index="0"/>
    <output id="CosY" datatype="Double" index="1"/>
    <output id="NegSinY" datatype="Double" index="2"/>
    <fbm id="YSinCosCalc" src="./SinCosCalc.xml"/>
  </block>

  <!-- Sine, Cosine, and Negative Sin of Z Rotation Angle -->
  <block id="SinCosZ">
    <input id="AngleZ" datatype="Double" index="0"/>
    <output id="SinZ" datatype="Double" index="0"/>
    <output id="CosZ" datatype="Double" index="1"/>
    <output id="NegSinZ" datatype="Double" index="2"/>
    <fbm id="ZSinCosCalc" src="./SinCosCalc.xml"/>
  </block>

  <!-- Calculation of Intermediate Matrix 1 -->
  <block id="I1">

    <!-- Scale Matrix -->
    <input id="Sc11" datatype="Double" index="0"/>
    <input id="Sc12" datatype="Double" index="1"/>
    <input id="Sc13" datatype="Double" index="2"/>
    <input id="Sc21" datatype="Double" index="3"/>
    <input id="Sc22" datatype="Double" index="4"/>
    <input id="Sc23" datatype="Double" index="5"/>
    <input id="Sc31" datatype="Double" index="6"/>
    <input id="Sc32" datatype="Double" index="7"/>
    <input id="Sc33" datatype="Double" index="8"/>

    <!-- Shear Matrix -->
    <input id="Sh11" datatype="Double" index="9"/>
    <input id="Sh12" datatype="Double" index="10"/>
    <input id="Sh13" datatype="Double" index="11"/>
    <input id="Sh21" datatype="Double" index="12"/>
    <input id="Sh22" datatype="Double" index="13"/>
    <input id="Sh23" datatype="Double" index="14"/>
    <input id="Sh31" datatype="Double" index="15"/>
    <input id="Sh32" datatype="Double" index="16"/>
    <input id="Sh33" datatype="Double" index="17"/>

    <!-- Intermediate Matrix 1 -->
    <output id="I1Out11" datatype="Double" index="0"/>
    <output id="I1Out12" datatype="Double" index="1"/>
    <output id="I1Out13" datatype="Double" index="2"/>
    <output id="I1Out21" datatype="Double" index="3"/>
    <output id="I1Out22" datatype="Double" index="4"/>
```

Figure D-3. Continued

```
    <output id="I1Out23" datatype="Double" index="5"/>
    <output id="I1Out31" datatype="Double" index="6"/>
    <output id="I1Out32" datatype="Double" index="7"/>
    <output id="I1Out33" datatype="Double" index="8"/>

    <!-- Matrix Multiplication 1 -->
    <fbm id="MatMult1" src="./MatMult.xml"/>
  </block>

  <!-- Calculation of Intermediate Matrix 2 -->
  <block id="I2">

    <!-- Intermediate Matrix 1 -->
    <input id="I1In11" datatype="Double" index="0"/>
    <input id="I1In12" datatype="Double" index="1"/>
    <input id="I1In13" datatype="Double" index="2"/>
    <input id="I1In21" datatype="Double" index="3"/>
    <input id="I1In22" datatype="Double" index="4"/>
    <input id="I1In23" datatype="Double" index="5"/>
    <input id="I1In31" datatype="Double" index="6"/>
    <input id="I1In32" datatype="Double" index="7"/>
    <input id="I1In33" datatype="Double" index="8"/>

    <!-- Z Rotation Matrix -->
    <input id="RotZ11" datatype="Double" index="9"/>
    <input id="RotZ12" datatype="Double" index="10"/>
    <input id="RotZ13" datatype="Double" index="11"/>
    <input id="RotZ21" datatype="Double" index="12"/>
    <input id="RotZ22" datatype="Double" index="13"/>
    <input id="RotZ23" datatype="Double" index="14"/>
    <input id="RotZ31" datatype="Double" index="15"/>
    <input id="RotZ32" datatype="Double" index="16"/>
    <input id="RotZ33" datatype="Double" index="17"/>

    <!-- Intermediate Matrix 2 -->
    <output id="I2Out11" datatype="Double" index="0"/>
    <output id="I2Out12" datatype="Double" index="1"/>
    <output id="I2Out13" datatype="Double" index="2"/>
    <output id="I2Out21" datatype="Double" index="3"/>
    <output id="I2Out22" datatype="Double" index="4"/>
    <output id="I2Out23" datatype="Double" index="5"/>
    <output id="I2Out31" datatype="Double" index="6"/>
    <output id="I2Out32" datatype="Double" index="7"/>
    <output id="I2Out33" datatype="Double" index="8"/>

    <!-- Matrix Multiplication 2 -->
    <fbm id="MatMult2" src="./MatMult.xml"/>
  </block>

  <!-- Calculation of Intermediate Matrix 3 -->
  <block id="I3">

    <!-- Intermediate Matrix 2 -->
    <input id="I2In11" datatype="Double" index="0"/>
    <input id="I2In12" datatype="Double" index="1"/>
    <input id="I2In13" datatype="Double" index="2"/>
    <input id="I2In21" datatype="Double" index="3"/>
    <input id="I2In22" datatype="Double" index="4"/>
    <input id="I2In23" datatype="Double" index="5"/>
    <input id="I2In31" datatype="Double" index="6"/>
    <input id="I2In32" datatype="Double" index="7"/>
```

Figure D-3. Continued

```
        <input id="I2In33" datatype="Double" index="8"/>

        <!-- Y Rotation Matrix -->
        <input id="RotY11" datatype="Double" index="9"/>
        <input id="RotY12" datatype="Double" index="10"/>
        <input id="RotY13" datatype="Double" index="11"/>
        <input id="RotY21" datatype="Double" index="12"/>
        <input id="RotY22" datatype="Double" index="13"/>
        <input id="RotY23" datatype="Double" index="14"/>
        <input id="RotY31" datatype="Double" index="15"/>
        <input id="RotY32" datatype="Double" index="16"/>
        <input id="RotY33" datatype="Double" index="17"/>

        <!-- Intermediate Matrix 3 -->
        <output id="I3Out11" datatype="Double" index="0"/>
        <output id="I3Out12" datatype="Double" index="1"/>
        <output id="I3Out13" datatype="Double" index="2"/>
        <output id="I3Out21" datatype="Double" index="3"/>
        <output id="I3Out22" datatype="Double" index="4"/>
        <output id="I3Out23" datatype="Double" index="5"/>
        <output id="I3Out31" datatype="Double" index="6"/>
        <output id="I3Out32" datatype="Double" index="7"/>
        <output id="I3Out33" datatype="Double" index="8"/>

        <!-- Matrix Multiplication 3 -->
        <fbm id="MatMult3" src="./MatMult.xml"/>
    </block>

    <!-- Calculation of Affine Matrix -->
    <block id="Affine">

        <!-- Intermediate Matrix 3 -->
        <input id="I3In11" datatype="Double" index="0"/>
        <input id="I3In12" datatype="Double" index="1"/>
        <input id="I3In13" datatype="Double" index="2"/>
        <input id="I3In21" datatype="Double" index="3"/>
        <input id="I3In22" datatype="Double" index="4"/>
        <input id="I3In23" datatype="Double" index="5"/>
        <input id="I3In31" datatype="Double" index="6"/>
        <input id="I3In32" datatype="Double" index="7"/>
        <input id="I3In33" datatype="Double" index="8"/>

        <!-- X Rotation Matrix -->
        <input id="RotX11" datatype="Double" index="9"/>
        <input id="RotX12" datatype="Double" index="10"/>
        <input id="RotX13" datatype="Double" index="11"/>
        <input id="RotX21" datatype="Double" index="12"/>
        <input id="RotX22" datatype="Double" index="13"/>
        <input id="RotX23" datatype="Double" index="14"/>
        <input id="RotX31" datatype="Double" index="15"/>
        <input id="RotX32" datatype="Double" index="16"/>
        <input id="RotX33" datatype="Double" index="17"/>

        <!-- Affine Matrix -->
        <output id="Affine11" datatype="Double" index="0"/>
        <output id="Affine12" datatype="Double" index="1"/>
        <output id="Affine13" datatype="Double" index="2"/>
        <output id="Affine21" datatype="Double" index="3"/>
        <output id="Affine22" datatype="Double" index="4"/>
        <output id="Affine23" datatype="Double" index="5"/>
        <output id="Affine31" datatype="Double" index="6"/>
```

Figure D-3. Continued

```
    <output id="Affine32" datatype="Double" index="7"/>
    <output id="Affine33" datatype="Double" index="8"/>

    <!-- Matrix Multiplication 4 -->
    <fbm id="MatMult4" src="./MatMult.xml"/>
  </block>

  <!-- Angles -->
  <trace from="Rotate_X" to="AngleX"/>
  <trace from="Rotate_Y" to="AngleY"/>
  <trace from="Rotate_Z" to="AngleZ"/>

  <!-- Scale Matrix -->
  <trace from="Scale_X" to="Sc11"/>
  <trace from="Const_Zero" to="Sc12"/>
  <trace from="Const_Zero" to="Sc13"/>
  <trace from="Const_Zero" to="Sc21"/>
  <trace from="Scale_Y" to="Sc22"/>
  <trace from="Const_Zero" to="Sc23"/>
  <trace from="Const_Zero" to="Sc31"/>
  <trace from="Const_Zero" to="Sc32"/>
  <trace from="Scale_Z" to="Sc33"/>

  <!-- Shear Matrix -->
  <trace from="Const_One" to="Sh11"/>
  <trace from="Shear_XY" to="Sh12"/>
  <trace from="Shear_XZ" to="Sh13"/>
  <trace from="Shear_YX" to="Sh21"/>
  <trace from="Const_One" to="Sh22"/>
  <trace from="Shear_YZ" to="Sh23"/>
  <trace from="Shear_ZX" to="Sh31"/>
  <trace from="Shear_ZY" to="Sh32"/>
  <trace from="Const_One" to="Sh33"/>

  <!-- Rotation Z Matrix -->
  <trace from="CosZ" to="RotZ11"/>
  <trace from="SinZ" to="RotZ12"/>
  <trace from="Const_Zero" to="RotZ13"/>
  <trace from="NegSinZ" to="RotZ21"/>
  <trace from="CosZ" to="RotZ22"/>
  <trace from="Const_Zero" to="RotZ23"/>
  <trace from="Const_Zero" to="RotZ31"/>
  <trace from="Const_Zero" to="RotZ32"/>
  <trace from="Const_One" to="RotZ33"/>

  <!-- Rotation Y Matrix -->
  <trace from="CosY" to="RotY11"/>
  <trace from="Const_Zero" to="RotY12"/>
  <trace from="NegSinY" to="RotY13"/>
  <trace from="Const_Zero" to="RotY21"/>
  <trace from="Const_One" to="RotY22"/>
  <trace from="Const_Zero" to="RotY23"/>
  <trace from="SinY" to="RotY31"/>
  <trace from="Const_Zero" to="RotY32"/>
  <trace from="CosY" to="RotY33"/>

  <!-- Rotation X Matrix -->
  <trace from="Const_One" to="RotX11"/>
  <trace from="Const_Zero" to="RotX12"/>
  <trace from="Const_Zero" to="RotX13"/>
  <trace from="Const_Zero" to="RotX21"/>
```

Figure D-3. Continued

```
      <trace from="CosX" to="RotX22"/>
      <trace from="SinX" to="RotX23"/>
      <trace from="Const_Zero" to="RotX31"/>
      <trace from="NegSinX" to="RotX32"/>
      <trace from="CosX" to="RotX33"/>

      <!-- Intermediate Matrix 1 -->
      <trace from="I1Out11" to="I1In11"/>
      <trace from="I1Out12" to="I1In12"/>
      <trace from="I1Out13" to="I1In13"/>
      <trace from="I1Out21" to="I1In21"/>
      <trace from="I1Out22" to="I1In22"/>
      <trace from="I1Out23" to="I1In23"/>
      <trace from="I1Out31" to="I1In31"/>
      <trace from="I1Out32" to="I1In32"/>
      <trace from="I1Out33" to="I1In33"/>

      <!-- Intermediate Matrix 2 -->
      <trace from="I2Out11" to="I2In11"/>
      <trace from="I2Out12" to="I2In12"/>
      <trace from="I2Out13" to="I2In13"/>
      <trace from="I2Out21" to="I2In21"/>
      <trace from="I2Out22" to="I2In22"/>
      <trace from="I2Out23" to="I2In23"/>
      <trace from="I2Out31" to="I2In31"/>
      <trace from="I2Out32" to="I2In32"/>
      <trace from="I2Out33" to="I2In33"/>

      <!-- Intermediate Matrix 3 -->
      <trace from="I3Out11" to="I3In11"/>
      <trace from="I3Out12" to="I3In12"/>
      <trace from="I3Out13" to="I3In13"/>
      <trace from="I3Out21" to="I3In21"/>
      <trace from="I3Out22" to="I3In22"/>
      <trace from="I3Out23" to="I3In23"/>
      <trace from="I3Out31" to="I3In31"/>
      <trace from="I3Out32" to="I3In32"/>
      <trace from="I3Out33" to="I3In33"/>

      <!-- Augmented Affine Transformation Matrix -->
      <trace from="Affine11" to="Augmented11"/>
      <trace from="Affine12" to="Augmented12"/>
      <trace from="Affine13" to="Augmented13"/>
      <trace from="Translate_X" to="Augmented14"/>
      <trace from="Affine21" to="Augmented21"/>
      <trace from="Affine22" to="Augmented22"/>
      <trace from="Affine23" to="Augmented23"/>
      <trace from="Translate_Y" to="Augmented24"/>
      <trace from="Affine31" to="Augmented31"/>
      <trace from="Affine32" to="Augmented32"/>
      <trace from="Affine33" to="Augmented33"/>
      <trace from="Translate_Z" to="Augmented34"/>
   </fbm>
</mxl:MXL>
```

Figure D-3. Continued

Sine and Cosine Calculation: SinCosCalc.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<mxl:MXL xmlns:mxl="http://www.cise.ufl.edu/~kdamkjer/rube/MXL"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.cise.ufl.edu/~kdamkjer/rube/MXL
http://www.cise.ufl.edu/~kdamkjer/rube/MXL/MXL.xsd">
   <input id="Angle" datatype="Double" index="0"/>
   <output id="Sin" datatype="Double" index="0"/>
   <output id="Cos" datatype="Double" index="1"/>
   <output id="NegSin" datatype="Double" index="2"/>
   <fbm id="SinCosCalc">

      <!-- Constant Negative One -->
      <block id="Negative">
         <output id="Const_Neg" datatype="Double" index="0"/>
         <script lang="JavaScript" src="./math.js" func="constant">
            <param index="0" value="-1"/>
         </script>
      </block>

      <!-- Sine of Input Angle -->
      <block id="SinAngle">
         <input id="AngleA" datatype="Double" index="0"/>
         <output id="SinA" datatype="Double" index="0"/>
         <script lang="JavaScript" src="./math.js" func="sin"/>
      </block>

      <!-- Cosine of Input Angle -->
      <block id="CosAngle">
         <input id="AngleB" datatype="Double" index="0"/>
         <output id="CosB" datatype="Double" index="0"/>
         <script lang="JavaScript" src="./math.js" func="cos"/>
      </block>

      <!-- Negative Sine of Input Angle -->
      <block id="NegSinAngle">
         <input id="PosSinC" datatype="Double" index="0"/>
         <input id="NegC" datatype="Double" index="1"/>
         <output id="NegSinC" datatype="Double" index="0"/>
         <script lang="JavaScript" src="./math.js" func="mult"/>
      </block>

      <!-- Angle In -->
      <trace from="Angle" to="AngleA"/>
      <trace from="Angle" to="AngleB"/>

      <!-- Negator -->
      <trace from="Const_Neg" to="NegC"/>
      <trace from="SinA" to="PosSinC"/>

      <!-- Sine Out -->
      <trace from="SinA" to="Sin"/>

      <!-- Cosine Out -->
      <trace from="CosB" to="Cos"/>

      <!-- Negative Sine Out -->
      <trace from="NegSinC" to="NegSin"/>
   </fbm>
</mxl:MXL>
```

Figure D-4. SinCosCalc.xml

3-by-3 by 3-by-3 Matrix Multiplication: MatMult.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<mxl:MXL xmlns:mxl="http://www.cise.ufl.edu/~kdamkjer/rube/MXL"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.cise.ufl.edu/~kdamkjer/rube/MXL
http://www.cise.ufl.edu/~kdamkjer/rube/MXL/MXL.xsd">

  <!-- Input Matrix A -->
  <input id="A11" datatype="Double" index="0"/>
  <input id="A12" datatype="Double" index="1"/>
  <input id="A13" datatype="Double" index="2"/>
  <input id="A21" datatype="Double" index="3"/>
  <input id="A22" datatype="Double" index="4"/>
  <input id="A23" datatype="Double" index="5"/>
  <input id="A31" datatype="Double" index="6"/>
  <input id="A32" datatype="Double" index="7"/>
  <input id="A33" datatype="Double" index="8"/>

  <!-- Input Matrix B -->
  <input id="B11" datatype="Double" index="9"/>
  <input id="B12" datatype="Double" index="10"/>
  <input id="B13" datatype="Double" index="11"/>
  <input id="B21" datatype="Double" index="12"/>
  <input id="B22" datatype="Double" index="13"/>
  <input id="B23" datatype="Double" index="14"/>
  <input id="B31" datatype="Double" index="15"/>
  <input id="B32" datatype="Double" index="16"/>
  <input id="B33" datatype="Double" index="17"/>

  <!-- Output Matrix C -->
  <output id="C11" datatype="Double" index="0"/>
  <output id="C12" datatype="Double" index="1"/>
  <output id="C13" datatype="Double" index="2"/>
  <output id="C21" datatype="Double" index="3"/>
  <output id="C22" datatype="Double" index="4"/>
  <output id="C23" datatype="Double" index="5"/>
  <output id="C31" datatype="Double" index="6"/>
  <output id="C32" datatype="Double" index="7"/>
  <output id="C33" datatype="Double" index="8"/>

  <!-- Matrix Multiplication FBM -->
  <fbm id="MatMult">

    <!-- Row-Column 1-1 Calculator -->
    <block id="RC11">

      <!-- Row-Column Input Vector A (Matrix A Row 1) -->
      <input id="RC11_A1" datatype="Double" index="0"/>
      <input id="RC11_A2" datatype="Double" index="1"/>
      <input id="RC11_A3" datatype="Double" index="2"/>

      <!-- Row-Column Input Vector B (Matrix B Column 1) -->
      <input id="RC11_B1" datatype="Double" index="3"/>
      <input id="RC11_B2" datatype="Double" index="4"/>
      <input id="RC11_B3" datatype="Double" index="5"/>

      <!-- Row-Column Output Vector C (Matrix C Entry 1-1) -->
      <output id="RC11_C1" datatype="Double" index="0"/>
```

Figure D-5. MatMult.xml

```
    <!-- Row-Column Calculator FBM -->
    <fbm id="RC11_RC" src="./RCCalc.xml"/>
  </block>

  <!-- Row-Column 1-1 Calculator -->
  <block id="RC12">

    <!-- Row-Column Input Vector A (Matrix A Row 1) -->
    <input id="RC12_A1" datatype="Double" index="0"/>
    <input id="RC12_A2" datatype="Double" index="1"/>
    <input id="RC12_A3" datatype="Double" index="2"/>

    <!-- Row-Column Input Vector B (Matrix B Column 2) -->
    <input id="RC12_B1" datatype="Double" index="3"/>
    <input id="RC12_B2" datatype="Double" index="4"/>
    <input id="RC12_B3" datatype="Double" index="5"/>

    <!-- Row-Column Output Vector C (Matrix C Entry 1-2) -->
    <output id="RC12_C1" datatype="Double" index="0"/>

    <!-- Row-Column Calculator FBM -->
    <fbm id="RC12_RC" src="./RCCalc.xml"/>
  </block>

  <!-- Row-Column 1-1 Calculator -->
  <block id="RC13">

    <!-- Row-Column Input Vector A (Matrix A Row 1) -->
    <input id="RC13_A1" datatype="Double" index="0"/>
    <input id="RC13_A2" datatype="Double" index="1"/>
    <input id="RC13_A3" datatype="Double" index="2"/>

    <!-- Row-Column Input Vector B (Matrix B Column 3) -->
    <input id="RC13_B1" datatype="Double" index="3"/>
    <input id="RC13_B2" datatype="Double" index="4"/>
    <input id="RC13_B3" datatype="Double" index="5"/>

    <!-- Row-Column Output Vector C (Matrix C Entry 1-3) -->
    <output id="RC13_C1" datatype="Double" index="0"/>

    <!-- Row-Column Calculator FBM -->
    <fbm id="RC13_RC" src="./RCCalc.xml"/>
  </block>

  <!-- Row-Column 2-1 Calculator -->
  <block id="RC21">

    <!-- Row-Column Input Vector A (Matrix A Row 2) -->
    <input id="RC21_A1" datatype="Double" index="0"/>
    <input id="RC21_A2" datatype="Double" index="1"/>
    <input id="RC21_A3" datatype="Double" index="2"/>

    <!-- Row-Column Input Vector B (Matrix B Column 1) -->
    <input id="RC21_B1" datatype="Double" index="3"/>
    <input id="RC21_B2" datatype="Double" index="4"/>
    <input id="RC21_B3" datatype="Double" index="5"/>

    <!-- Row-Column Output Vector C (Matrix C Entry 2-1) -->
    <output id="RC21_C1" datatype="Double" index="0"/>

    <!-- Row-Column Calculator FBM -->
```

Figure D-5. Continued

```
    <fbm id="RC21_RC" src="./RCCalc.xml"/>
  </block>

  <!-- Row-Column 2-2 Calculator -->
  <block id="RC22">

    <!-- Row-Column Input Vector A (Matrix A Row 2) -->
    <input id="RC22_A1" datatype="Double" index="0"/>
    <input id="RC22_A2" datatype="Double" index="1"/>
    <input id="RC22_A3" datatype="Double" index="2"/>

    <!-- Row-Column Input Vector B (Matrix B Column 2) -->
    <input id="RC22_B1" datatype="Double" index="3"/>
    <input id="RC22_B2" datatype="Double" index="4"/>
    <input id="RC22_B3" datatype="Double" index="5"/>

    <!-- Row-Column Output Vector C (Matrix C Entry 2-2) -->
    <output id="RC22_C1" datatype="Double" index="0"/>

    <!-- Row-Column Calculator FBM -->
    <fbm id="RC22_RC" src="./RCCalc.xml"/>
  </block>

  <!-- Row-Column 2-3 Calculator -->
  <block id="RC23">

    <!-- Row-Column Input Vector A (Matrix A Row 2) -->
    <input id="RC23_A1" datatype="Double" index="0"/>
    <input id="RC23_A2" datatype="Double" index="1"/>
    <input id="RC23_A3" datatype="Double" index="2"/>

    <!-- Row-Column Input Vector B (Matrix B Column 3) -->
    <input id="RC23_B1" datatype="Double" index="3"/>
    <input id="RC23_B2" datatype="Double" index="4"/>
    <input id="RC23_B3" datatype="Double" index="5"/>

    <!-- Row-Column Output Vector C (Matrix C Entry 2-3) -->
    <output id="RC23_C1" datatype="Double" index="0"/>

    <!-- Row-Column Calculator FBM -->
    <fbm id="RC23_RC" src="./RCCalc.xml"/>
  </block>

  <!-- Row-Column 3-1 Calculator -->
  <block id="RC31">

    <!-- Row-Column Input Vector A (Matrix A Row 3) -->
    <input id="RC31_A1" datatype="Double" index="0"/>
    <input id="RC31_A2" datatype="Double" index="1"/>
    <input id="RC31_A3" datatype="Double" index="2"/>

    <!-- Row-Column Input Vector B (Matrix B Column 1) -->
    <input id="RC31_B1" datatype="Double" index="3"/>
    <input id="RC31_B2" datatype="Double" index="4"/>
    <input id="RC31_B3" datatype="Double" index="5"/>

    <!-- Row-Column Output Vector C (Matrix C Entry 3-1) -->
    <output id="RC31_C1" datatype="Double" index="0"/>

    <!-- Row-Column Calculator FBM -->
    <fbm id="RC31_RC" src="./RCCalc.xml"/>
```

Figure D-5. Continued

```
</block>

<!-- Row-Column 3-2 Calculator -->
<block id="RC32">

   <!-- Row-Column Input Vector A (Matrix A Row 3) -->
   <input id="RC32_A1" datatype="Double" index="0"/>
   <input id="RC32_A2" datatype="Double" index="1"/>
   <input id="RC32_A3" datatype="Double" index="2"/>

   <!-- Row-Column Input Vector B (Matrix B Column 2) -->
   <input id="RC32_B1" datatype="Double" index="3"/>
   <input id="RC32_B2" datatype="Double" index="4"/>
   <input id="RC32_B3" datatype="Double" index="5"/>

   <!-- Row-Column Output Vector C (Matrix C Entry 3-2) -->
   <output id="RC32_C1" datatype="Double" index="0"/>

   <!-- Row-Column Calculator FBM -->
   <fbm id="RC32_RC" src="./RCCalc.xml"/>
</block>

<!-- Row-Column 3-3 Calculator -->
<block id="RC33">

   <!-- Row-Column Input Vector A (Matrix A Row 3) -->
   <input id="RC33_A1" datatype="Double" index="0"/>
   <input id="RC33_A2" datatype="Double" index="1"/>
   <input id="RC33_A3" datatype="Double" index="2"/>

   <!-- Row-Column Input Vector B (Matrix B Column 3) -->
   <input id="RC33_B1" datatype="Double" index="3"/>
   <input id="RC33_B2" datatype="Double" index="4"/>
   <input id="RC33_B3" datatype="Double" index="5"/>

   <!-- Row-Column Output Vector C (Matrix C Entry 3-3) -->
   <output id="RC33_C1" datatype="Double" index="0"/>

   <!-- Row-Column Calculator FBM -->
   <fbm id="RC33_RC" src="./RCCalc.xml"/>
</block>

<!-- Traces to calculate Matrix C Entry 1-1) -->
<trace from="A11" to="RC11_A1"/>
<trace from="A12" to="RC11_A2"/>
<trace from="A13" to="RC11_A3"/>
<trace from="B11" to="RC11_B1"/>
<trace from="B21" to="RC11_B2"/>
<trace from="B31" to="RC11_B3"/>
<trace from="RC11_C1" to="C11"/>

<!-- Traces to calculate Matrix C Entry 1-2) -->
<trace from="A11" to="RC12_A1"/>
<trace from="A12" to="RC12_A2"/>
<trace from="A13" to="RC12_A3"/>
<trace from="B12" to="RC12_B1"/>
<trace from="B22" to="RC12_B2"/>
<trace from="B32" to="RC12_B3"/>
<trace from="RC12_C1" to="C12"/>

<!-- Traces to calculate Matrix C Entry 1-3) -->
```

Figure D-5. Continued

```
<trace from="A11" to="RC13_A1"/>
<trace from="A12" to="RC13_A2"/>
<trace from="A13" to="RC13_A3"/>
<trace from="B13" to="RC13_B1"/>
<trace from="B23" to="RC13_B2"/>
<trace from="B33" to="RC13_B3"/>
<trace from="RC13_C1" to="C13"/>

<!-- Traces to calculate Matrix C Entry 2-1) -->
<trace from="A21" to="RC21_A1"/>
<trace from="A22" to="RC21_A2"/>
<trace from="A23" to="RC21_A3"/>
<trace from="B11" to="RC21_B1"/>
<trace from="B21" to="RC21_B2"/>
<trace from="B31" to="RC21_B3"/>
<trace from="RC21_C1" to="C21"/>

<!-- Traces to calculate Matrix C Entry 2-2) -->
<trace from="A21" to="RC22_A1"/>
<trace from="A22" to="RC22_A2"/>
<trace from="A23" to="RC22_A3"/>
<trace from="B12" to="RC22_B1"/>
<trace from="B22" to="RC22_B2"/>
<trace from="B32" to="RC22_B3"/>
<trace from="RC22_C1" to="C22"/>

<!-- Traces to calculate Matrix C Entry 2-3) -->
<trace from="A21" to="RC23_A1"/>
<trace from="A22" to="RC23_A2"/>
<trace from="A23" to="RC23_A3"/>
<trace from="B13" to="RC23_B1"/>
<trace from="B23" to="RC23_B2"/>
<trace from="B33" to="RC23_B3"/>
<trace from="RC23_C1" to="C23"/>

<!-- Traces to calculate Matrix C Entry 3-1) -->
<trace from="A31" to="RC31_A1"/>
<trace from="A32" to="RC31_A2"/>
<trace from="A33" to="RC31_A3"/>
<trace from="B11" to="RC31_B1"/>
<trace from="B21" to="RC31_B2"/>
<trace from="B31" to="RC31_B3"/>
<trace from="RC31_C1" to="C31"/>

<!-- Traces to calculate Matrix C Entry 3-2) -->
<trace from="A31" to="RC32_A1"/>
<trace from="A32" to="RC32_A2"/>
<trace from="A33" to="RC32_A3"/>
<trace from="B12" to="RC32_B1"/>
<trace from="B22" to="RC32_B2"/>
<trace from="B32" to="RC32_B3"/>
<trace from="RC32_C1" to="C32"/>

<!-- Traces to calculate Matrix C Entry 3-3) -->
<trace from="A31" to="RC33_A1"/>
<trace from="A32" to="RC33_A2"/>
<trace from="A33" to="RC33_A3"/>
<trace from="B13" to="RC33_B1"/>
<trace from="B23" to="RC33_B2"/>
<trace from="B33" to="RC33_B3"/>
<trace from="RC33_C1" to="C33"/>
```

Figure D-5. Continued

```
  </fbm>
</mxl:MXL>
```

Figure D-5. Continued

Row-Column Vector Multiplication: RCCalc.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<mxl:MXL xmlns:mxl="http://www.cise.ufl.edu/~kdamkjer/rube/MXL"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.cise.ufl.edu/~kdamkjer/rube/MXL
http://www.cise.ufl.edu/~kdamkjer/rube/MXL/MXL.xsd">

  <!-- Input Vector A -->
  <input id="A1" datatype="Double" index="0"/>
  <input id="A2" datatype="Double" index="1"/>
  <input id="A3" datatype="Double" index="2"/>

  <!-- Input Vector B -->
  <input id="B1" datatype="Double" index="3"/>
  <input id="B2" datatype="Double" index="4"/>
  <input id="B3" datatype="Double" index="5"/>

  <!-- Output Scalar C -->
  <output id="C" datatype="Double" index="0"/>

  <!-- Row-Column Calculator FBM -->
  <fbm id="RCCalc">

    <!-- Multiplier 1 -->
    <block id="AB1Mult">
      <input id="AB1Mult_Factor1" datatype="Double" index="0"/>
      <input id="AB1Mult_Factor2" datatype="Double" index="1"/>
      <output id="AB1Mult_Product" datatype="Double" index="0"/>
      <script lang="JavaScript" src="./math.js" func="mult"/>
    </block>

    <!-- Multiplier 2 -->
    <block id="AB2Mult">
      <input id="AB2Mult_Factor1" datatype="Double" index="0"/>
      <input id="AB2Mult_Factor2" datatype="Double" index="1"/>
      <output id="AB2Mult_Product" datatype="Double" index="0"/>
      <script lang="JavaScript" src="./math.js" func="mult"/>
    </block>

    <!-- Multiplier 3 -->
    <block id="AB3Mult">
      <input id="AB3Mult_Factor1" datatype="Double" index="0"/>
      <input id="AB3Mult_Factor2" datatype="Double" index="1"/>
      <output id="AB3Mult_Product" datatype="Double" index="0"/>
      <script lang="JavaScript" src="./math.js" func="mult"/>
    </block>

    <!-- Adder 1 -->
    <block id="Mult12Adder">
      <input id="Mult12Adder_Addend1" datatype="Double" index="0"/>
      <input id="Mult12Adder_Addend2" datatype="Double" index="1"/>
      <output id="Mult12Adder_Sum" datatype="Double" index="0"/>
```

Figure D-6. RCCalc.xml

```
        <script lang="JavaScript" src="./math.js" func="add"/>
      </block>

      <!-- Adder 2 -->
      <block id="Mult123Adder">
        <input id="Mult123Adder_Addend1" datatype="Double" index="0"/>
        <input id="Mult123Adder_Addend2" datatype="Double" index="1"/>
        <output id="Mult123Adder_Sum" datatype="Double" index="0"/>
        <script lang="JavaScript" src="./math.js" func="add"/>
      </block>

      <!-- Inputs to Multiplier 1 -->
      <trace from="A1" to="AB1Mult_Factor1"/>
      <trace from="B1" to="AB1Mult_Factor2"/>

      <!-- Inputs to Multiplier 2 -->
      <trace from="A2" to="AB2Mult_Factor1"/>
      <trace from="B2" to="AB2Mult_Factor2"/>

      <!-- Inputs to Multiplier 3 -->
      <trace from="A3" to="AB3Mult_Factor1"/>
      <trace from="B3" to="AB3Mult_Factor2"/>

      <!-- Inputs to Adder 1 -->
      <trace from="AB1Mult_Product" to="Mult12Adder_Addend1"/>
      <trace from="AB2Mult_Product" to="Mult12Adder_Addend2"/>

      <!-- Inputs to Adder 2 -->
      <trace from="Mult12Adder_Sum" to="Mult123Adder_Addend1"/>
      <trace from="AB3Mult_Product" to="Mult123Adder_Addend2"/>

      <!-- FBM Output -->
      <trace from="Mult123Adder_Sum" to="C"/>
    </fbm>
</mxl:MXL>
```

Figure D-6. Continued

IFS Attractor Point Calculation: PointCalc.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<mxl:MXL xmlns:mxl="http://www.cise.ufl.edu/~kdamkjer/rube/MXL"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.cise.ufl.edu/~kdamkjer/rube/MXL
http://www.cise.ufl.edu/~kdamkjer/rube/MXL/MXL.xsd">

    <!-- Current Point -->
    <input id="CurrentX" datatype="Double" index="0"/>
    <input id="CurrentY" datatype="Double" index="1"/>
    <input id="CurrentZ" datatype="Double" index="2"/>

    <!-- Augmented Affine Matrix -->
    <input id="Augmented11" datatype="Double" index="3"/>
    <input id="Augmented12" datatype="Double" index="4"/>
    <input id="Augmented13" datatype="Double" index="5"/>
    <input id="Augmented14" datatype="Double" index="6"/>
    <input id="Augmented21" datatype="Double" index="7"/>
    <input id="Augmented22" datatype="Double" index="8"/>
```

Figure D-7. PointCalc.xml

```
<input id="Augmented23" datatype="Double" index="9"/>
<input id="Augmented24" datatype="Double" index="10"/>
<input id="Augmented31" datatype="Double" index="11"/>
<input id="Augmented32" datatype="Double" index="12"/>
<input id="Augmented33" datatype="Double" index="13"/>
<input id="Augmented34" datatype="Double" index="14"/>

<!-- New Point -->
<output id="NewX" datatype="Double" index="0"/>
<output id="NewY" datatype="Double" index="1"/>
<output id="NewZ" datatype="Double" index="2"/>

<!-- New Point Calculation FBM -->
<fbm id="PointCalc">

   <!-- Calculate the New X -->
   <block id="XTransform">
      <input id="a" datatype="Double" index="0"/>
      <input id="b" datatype="Double" index="1"/>
      <input id="c" datatype="Double" index="2"/>
      <input id="xx" datatype="Double" index="3"/>
      <input id="xy" datatype="Double" index="4"/>
      <input id="xz" datatype="Double" index="5"/>
      <output id="tempXOut" datatype="Double" index="0"/>
      <fbm id="XTransformation" src="./RcCalc.xml"/>
   </block>
   <block id="XTranslate">
      <input id="tempXIn" datatype="Double" index="0"/>
      <input id="j" datatype="Double" index="1"/>
      <output id="XOut" datatype="Double" index="0"/>
      <script lang="JavaScript" src="./math" func="add"/>
   </block>

   <!-- Calculate the New Y -->
   <block id="YTransform">
      <input id="d" datatype="Double" index="0"/>
      <input id="e" datatype="Double" index="1"/>
      <input id="f" datatype="Double" index="2"/>
      <input id="yx" datatype="Double" index="3"/>
      <input id="yy" datatype="Double" index="4"/>
      <input id="yz" datatype="Double" index="5"/>
      <output id="tempYOut" datatype="Double" index="0"/>
      <fbm id="YTransformation" src="./RcCalc.xml"/>
   </block>
   <block id="YTranslate">
      <input id="tempYIn" datatype="Double" index="0"/>
      <input id="k" datatype="Double" index="1"/>
      <output id="YOut" datatype="Double" index="0"/>
      <script lang="JavaScript" src="./math" func="add"/>
   </block>

   <!-- Calculate the New Z -->
   <block id="ZTransform">
      <input id="g" datatype="Double" index="0"/>
      <input id="h" datatype="Double" index="1"/>
      <input id="i" datatype="Double" index="2"/>
      <input id="zx" datatype="Double" index="3"/>
      <input id="zy" datatype="Double" index="4"/>
      <input id="zz" datatype="Double" index="5"/>
      <output id="tempZOut" datatype="Double" index="0"/>
      <fbm id="ZTransformation" src="./RcCalc.xml"/>
```

Figure D-7. Continued

```
    </block>
    <block id="ZTranslate">
      <input id="tempZIn" datatype="Double" index="0"/>
      <input id="l" datatype="Double" index="1"/>
      <output id="ZOut" datatype="Double" index="0"/>
      <script lang="JavaScript" src="./math" func="add"/>
    </block>

    <!-- Connect Blocks to calculate New X -->
    <trace from="Augmented11" to="a"/>
    <trace from="Augmented12" to="b"/>
    <trace from="Augmented13" to="c"/>
    <trace from="Augmented14" to="j"/>
    <trace from="CurrentX" to="xx"/>
    <trace from="CurrentY" to="xy"/>
    <trace from="CurrentZ" to="xz"/>
    <trace from="tempXOut" to="tempXIn"/>
    <trace from="XOut" to="NewX"/>

    <!-- Connect Blocks to calculate New Y -->
    <trace from="Augmented21" to="d"/>
    <trace from="Augmented22" to="e"/>
    <trace from="Augmented23" to="f"/>
    <trace from="Augmented24" to="k"/>
    <trace from="CurrentX" to="yx"/>
    <trace from="CurrentY" to="yy"/>
    <trace from="CurrentZ" to="yz"/>
    <trace from="tempYOut" to="tempYIn"/>
    <trace from="YOut" to="NewY"/>

    <!-- Connect Blocks to calculate New Z -->
    <trace from="Augmented31" to="g"/>
    <trace from="Augmented32" to="h"/>
    <trace from="Augmented33" to="i"/>
    <trace from="Augmented34" to="l"/>
    <trace from="CurrentX" to="zx"/>
    <trace from="CurrentY" to="zy"/>
    <trace from="CurrentZ" to="zz"/>
    <trace from="tempZOut" to="tempZIn"/>
    <trace from="ZOut" to="NewZ"/>
  </fbm>
</mxl:MXL>
```

Figure D-7. Continued

IFS-Specific Semantics JavaScript: Barnsley.js

```
// Barnsley.js contains functions for driving IFS fractal generation
// models that use a set of four potential transformations.

/*************************************************************************\
 * function: display()                                                   *
 *-----------------------------------------------------------------------*
 * Displays the New Point data output by the NewPoint block.             *
\*************************************************************************/
function display() {
  print(this.input[0]);
  print(this.input[1]);
```

Figure D-8. Barnsley.js

```
  print(this.input[2]);
}

/*************************************************************************\
 * function: stemGen()                                                   *
 *-----------------------------------------------------------------------*
 * Generates transform data for the Stem state of the IFS.               *
\*************************************************************************/
function stemGen() {
  this.output[0]  =  0.00; // Scale X to  nothing
  this.output[1]  =  0.16; // Scale Y to  16%
  this.output[2]  =  0.00; // Scale Z to  nothing

  this.output[3]  =  0.00; // Shear X along Y by  0.00 units
  this.output[4]  =  0.00; // Shear X along Z by  0.00 units
  this.output[5]  =  0.00; // Shear Y along X by  0.00 units
  this.output[6]  =  0.00; // Shear Y along Z by  0.00 units
  this.output[7]  =  0.00; // Shear Z along X by  0.00 units
  this.output[8]  =  0.00; // Shear Z along Y by  0.00 units

  this.output[9]  =  0.00; // Rotate about X by  0.00 degrees
  this.output[10] =  0.00; // Rotate about Y by  0.00 degrees
  this.output[11] =  0.00; // Rotate about Z by  0.00 degrees

  this.output[12] =  0.00; // Translate along X by  0.00 units
  this.output[13] =  0.00; // Translate along Y by  0.00 units
  this.output[14] =  0.00; // Translate along Z by  0.00 units
}

/*************************************************************************\
 * function: leftGen()                                                   *
 *-----------------------------------------------------------------------*
 * Generates transform data for the Stem state of the IFS.               *
\*************************************************************************/
function leftGen() {
  this.output[0]  =  0.30; // Scale X to 30%
  this.output[1]  =  0.37; // Scale Y to 37%
  this.output[2]  =  0.00; // Scale Z to nothing

  this.output[3]  =  0.00; // Shear X along Y by  0.00 units
  this.output[4]  =  0.00; // Shear X along Z by  0.00 units
  this.output[5]  =  0.00; // Shear Y along X by  0.00 units
  this.output[6]  =  0.00; // Shear Y along Z by  0.00 units
  this.output[7]  =  0.00; // Shear Z along X by  0.00 units
  this.output[8]  =  0.00; // Shear Z along Y by  0.00 units

  this.output[9]  =  0.00; // Rotate about X by  0.00 degrees
  this.output[10] =  0.00; // Rotate about Y by  0.00 degrees
  this.output[11] = 50.00; // Rotate about Z by 50.00 degrees

  this.output[12] =  0.00; // Translate along X by  0.00 units
  this.output[13] =  0.44; // Translate along Y by  0.44 units
  this.output[14] =  0.00; // Translate along Z by  0.00 units
}

/*************************************************************************\
 * function: rightGen()                                                  *
 *-----------------------------------------------------------------------*
 * Generates transform data for the Stem state of the IFS.               *
```

Figure D-8. Continued

```
\*************************************************************************/
function rightGen() {
  this.output[0]  =  0.30; // Scale X to 30%
  this.output[1]  = -0.34; // Scale Y to 34% and reflect about Y axis
  this.output[2]  =  0.00; // Scale Z to nothing

  this.output[3]  =  0.00; // Shear X along Y by  0.00 units
  this.output[4]  =  0.00; // Shear X along Z by  0.00 units
  this.output[5]  =  0.00; // Shear Y along X by  0.00 units
  this.output[6]  =  0.00; // Shear Y along Z by  0.00 units
  this.output[7]  =  0.00; // Shear Z along X by  0.00 units
  this.output[8]  =  0.00; // Shear Z along Y by  0.00 units

  this.output[9]  =  0.00; // Rotate about X by  0.00 degrees
  this.output[10] =  0.00; // Rotate about Y by  0.00 degrees
  this.output[11] = 49.00; // Rotate about Z by 49.00 degrees

  this.output[12] =  0.00; // Translate along X by  0.00 units
  this.output[13] =  1.60; // Translate along Y by  1.60 units
  this.output[14] =  0.00; // Translate along Z by  0.00 units
}

/*************************************************************************\
 * function: mainGen()                                                   *
 *-----------------------------------------------------------------------*
 * Generates transform data for the Stem state of the IFS.               *
\*************************************************************************/
function mainGen() {
  this.output[0]  =  0.85; // Scale X to 85%
  this.output[1]  =  0.85; // Scale Y to 85%
  this.output[2]  =  0.00; // Scale Z to nothing

  this.output[3]  =  0.00; // Shear X along Y by  0.00 units
  this.output[4]  =  0.00; // Shear X along Z by  0.00 units
  this.output[5]  =  0.00; // Shear Y along X by  0.00 units
  this.output[6]  =  0.00; // Shear Y along Z by  0.00 units
  this.output[7]  =  0.00; // Shear Z along X by  0.00 units
  this.output[8]  =  0.00; // Shear Z along Y by  0.00 units

  this.output[9]  =  7.00; // Rotate about X by  7.00 degrees
  this.output[10] =  0.00; // Rotate about Y by  0.00 degrees
  this.output[11] = -2.50; // Rotate about Z by -2.50 degrees

  this.output[12] =  0.00; // Translate along X by  0.00 units
  this.output[13] =  1.60; // Translate along Y by  1.60 units
  this.output[14] =  0.00; // Translate along Z by  0.00 units
}

/*************************************************************************\
 * function: toStem()                                                    *
 *-----------------------------------------------------------------------*
 * Transitions to the Stem state fire 2% of the time.  This is achieved  *
 * by setting the transition to fire when the random number input (a     *
 * value between 0 and 1 inclusive) is less than or equal to 0.020.      *
\*************************************************************************/
function toStem() {
  if (parseFloat(this.input[0]) <= 0.020) {
    this.output[0] = true;
  } else {
```

Figure D-8. Continued

```
    this.output[0] = false;
  }
}

/*************************************************************************\
 * function: toLeftFrond()                                               *
 *-----------------------------------------------------------------------*
 * Transitions to the LeftFrond state fire 11.5% of the time.  This is   *
 * achieved by setting the transition to fire when the random number     *
 * input (a value between 0 and 1 inclusive) is between 0.020 and 0.135. *
\*************************************************************************/
function toLeftFrond() {
  if (parseFloat(this.input[0]) > 0.020 &&
      parseFloat(this.input[0]) <= 0.135) {
    this.output[0] = true;
  } else {
    this.output[0] = false;
  }
}

/*************************************************************************\
 * function: toRightFrond()                                              *
 *-----------------------------------------------------------------------*
 * Transitions to the RightFrond state fire 11.5% of the time.  This is  *
 * achieved by setting the transition to fire when the random number     *
 * input (a value between 0 and 1 inclusive) is between 0.135 and 0.250. *
\*************************************************************************/
function toRightFrond() {
  if (parseFloat(this.input[0]) > 0.135 &&
      parseFloat(this.input[0]) <= 0.250) {
    this.output[0] = true;
  } else {
    this.output[0] = false;
  }
}

/*************************************************************************\
 * function: toMainFrond()                                               *
 *-----------------------------------------------------------------------*
 * Transitions to the MainFrond state fire 75% of the time.  This is     *
 * achieved by setting the transition to fire when the random number     *
 * input (a value between 0 and 1 inclusive) is greater than 0.250.      *
\*************************************************************************/
function toMainFrond() {
  if (parseFloat(this.input[0]) > 0.250) {
    this.output[0] = true;
  } else {
    this.output[0] = false;
  }
}
```

Figure D-8. Continued

RUBE Math Library Functions JavaScript: math.js

```
// math.js is a very basic mathematics library for rube(tm) it will grow
// as more functions are needed.

/*************************************************************************\
 * function: constant()                                                  *
 *-----------------------------------------------------------------------*
 * Returns the constant specified by parameter.                          *
\*************************************************************************/
function constant(value) {
  this.output[0] = value;
}

/*************************************************************************\
 * function: rand()                                                      *
 *-----------------------------------------------------------------------*
 * Returns a random number between 0 and 1 inclusive.                    *
\*************************************************************************/
function rand() {
  this.output[0] = Math.random();
}

/*************************************************************************\
 * function: add()                                                       *
 *-----------------------------------------------------------------------*
 * Returns the sum of two inputs.                                        *
\*************************************************************************/
function add() {
  this.output[0] = this.input[0] + this.input[1];
}

/*************************************************************************\
 * function: mult()                                                      *
 *-----------------------------------------------------------------------*
 * Returns the product of two inputs.                                    *
\*************************************************************************/
function mult() {
  this.output[0] = this.input[0] * this.input[1];
}

/*************************************************************************\
 * function: sin()                                                       *
 *-----------------------------------------------------------------------*
 * Returns the sin of the angle (in degrees) specified by the input.     *
\*************************************************************************/
function sin() {
  this.output[0] = Math.sin(this.input[0] * Math.PI / 180);
}

/*************************************************************************\
 * function: cos()                                                       *
 *-----------------------------------------------------------------------*
 * Returns the cosine of the angle (in degrees) specified by the input.  *
\*************************************************************************/
function cos() {
  this.output[0] = Math.cos(this.input[0] * Math.PI / 180);
}
```

Figure D-9. math.js

LIST OF REFERENCES

1. Black, John B., and Gordon H. Bower. "Episodes as Chunks in Narrative Memory." *Journal of Verbal Learning and Verbal Behaviour* 18.3 (1979): 309-318.

2. Bruner, Jerome. *Actual Minds, Possible Worlds*. London: Harvard University Press, 1986.

3. Bussey, Janice. "Mathematics for the Alternative High School Student." *The Journal of Court, Community, and Alternative Schools* 14 (2001): 45-51.

4. Clements, Douglas H., and Sue McMillen. "Rethinking 'Concrete' Manipulatives." *Teaching Children Mathematics* 2.5 (1996): 270-279.

5. Damkjer, Kristian Linn. *RUBE*. *Department of Computer and Information Science and Engineering*. University of Florida. 29 January 2003. ‹http://www.cise.ufl.edu/~kdamkjer/rube›.

6. Douglas, Yellowlees, and Andrew Hargadon. *The Pleasure Principle: Immersion, Engagement, Flow*. Proceedings of the eleventh ACM on Hypertext and Hypermedia, 30 May–3 June 2000, San Antonio. New York: ACM Press, 2000.

7. Eppstein, David. "Finite Automata and String Matching." *ICS 161, Winter 2003: Design and Analysis of Algorithms*. *School of Information and Computer Science*. University of California at Irvine. 22 February 1996. ‹http://www.ics.uci.edu/~eppstein/161/960222.html›.

8. Fishwick, Paul. *Simulation Model Design and Execution: Building Digital Worlds*. Englewood Cliffs: Prentice Hall, 1995.

9. Fishwick, Paul. "Integrating Continuous and Discrete Models with Object Oriented Physical Modeling." Paper presented at 1997 Western Simulation Multiconference, Phoenix, Arizona, 1997.

10. Fishwick, Paul. *Using XML for Simulation Modeling*. Proceedings of the 2002 Winter Simulation Conference, 8–11 December 2002, San Diego. Ed. E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. Charnes. Piscataway: Institute of Electrical and Electronic Engineers, 2002.

11. Fishwick, Paul. "Aesthetic Computing: Making Artistic Mathematics & Software." *YLEM Journal* 22.10 (2002): 6-11.

12. Fishwick, Paul. *RUBE*. *Department of Computer and Information Science and Engineering*. University of Florida. 11 September 2002. ‹http://www.cise.ufl.edu/~fishwick/rube›.

13. Hughes, Bob. *Dust or Magic:Secrets of Successful Multimedia Design*. London: Addison-Wesley, 2000.

14. Kim, Taewoo. "A 3D XML-Based Modeling and Simulation Framework for Dynamic Models." Doctoral Dissertation, University of Florida, 2002.

15. Lakoff, George with Jane Espenson and Alan Schwartz. *Master Metaphor List*, 2nd ed. University of California at Berkeley: Cognitive Linguistics Group, 1991.

16. Park, Minho. "Simpack J/S: A Web-Oriented Toolkit for Discrete Event Simulation." Master's Thesis, University of Florida, 2002.

17. Schmandt-Besserat, Denise. *How Writing Came About*. Austin: University of Texas Press, 1997.

BIOGRAPHICAL SKETCH

Kristian Linn Damkjer was born on March 6th, 1980, in Jacksonville, Florida. He received an International Baccalaureate diploma from Stanton College Preparatory School in Jacksonville, Florida, in 1998 and was accepted into the University of Florida to pursue higher education. In 2001, he received his Bachelor of Science degree from the University of Florida in computer and information science through the College of Liberal Arts and Sciences. That same year he was again accepted at the University of Florida, this time as a member of the inaugural class of digital arts and sciences master's students through the College of Engineering. He will receive his Master of Science in computer engineering with a specialization in digital arts and sciences in May of 2003.